Diseño y análisis arquitectónico de dos posibles implementaciones de una red social

Grado en ingeniería informática

(Ingeniería de software)

Guillermo Valentín Pérez

Director: Ernest Teniente

26 de Junio de 2019

Resumen

Las arquitecturas basadas en microservicios han ganado mucha popularidad estos últimos años, y es que en la actualidad, la mitad de las empresas ya las utilizan para alguna de sus aplicaciones. Los beneficios que ofrecen respecto a los denominados sistemas monolíticos son muchos y ya conocidos, pero también lo son sus desventajas.

El objetivo de este trabajo es el de desarrollar una red social haciendo uso de estos dos estilos arquitectónicos con el fin de ganar conocimiento acerca de las herramientas y bibliotecas necesarias para su implementación así como de los aspectos más importantes a tener en cuenta durante su diseño. Una vez finalizados, dichos sistemas serán probados para medir la diferencia de rendimiento entre ambos y poder determinar en que casos debería usarse uno u otro dependiendo de los requisitos del sistema.

Resum

Les arquitectures basades en microserveis han guanyat molta popularitat en els últims anys, i és que en l'actualitat, la meitat de les empreses ja les fan servir per a alguna de les seves aplicacions. Els beneficis que ofereixen respecte als anomenats sistemes monolítics son molts i ja coneguts, però també ho són els seus desavantatges.

L'objectiu d'aquest treball és el de desenvolupar una xarxa social fent ús d'aquests dos estils arquitectònics amb la finalitat de guanyar coneixement sobre les eines i llibreries necessàries per a la seva implementació així com dels aspectes més importants a tenir en compte durant el seu disseny. Una vegada finalitzats, aquests sistemes seran provats per a mesurar la diferència de rendiment entre tots dos i poder determinar en que casos hauria d'usar-se l'un o l'altre depenent dels requisits del sistema.

Abstract

Microservice-based architectures have gained much popularity in recent years, and today, half of companies already use them for some of their applications. The benefits they offer regarding the so-called monolithic systems are many and well-known, but so are their disadvantages.

The objective of this work is to develop a social network using these two architectural styles in order to gain knowledge about the tools and libraries necessary for its implementation as well as the most important aspects to be taken into account during its design. Once completed, these systems will be tested in order to measure the difference in performance between the two and to determine in which cases one or the other should be used depending on the requirements of the system.

Agradecimientos

Quisiera expresar mi más sincero agradecimiento a todos aquellos profesores que, a lo largo de todos estos años, me han transmitido no solo sus conocimientos sino también su pasión por aquello que enseñaban.

Índice

Índice de tablas

Índice de ilustraciones

Glosario

API Acrónimo de application programming interface. Consiste en un conjunto de subrutinas, funciones y procedimientos que ofrece cierta biblioteca para ser utilizado por otro software como una capa de abstracción.

AWS Acrónimo de Amazon Web Services. Consiste en una colección de servicios web que en conjunto forman una plataforma de computación en la nube, ofrecida por Amazon.com.

Back-end Parte del software que procesa la entrada desde el front-end.

Backlog Modelo de trabajo a realizar que contiene una lista ordenada de requisitos que un equipo de scrum mantiene para un producto.

Bean En Spring, los objetos que forman la columna vertebral de una aplicación y que son administrados por el contenedor Spring se denominan beans. Estos son instanciados, ensamblados y gestionados por el framework.

Biblioteca En informática, una biblioteca o, llamada por vicio del lenguaje librería (del inglés library) es un conjunto de implementaciones funcionales, codificadas en un lenguaje de programación, que ofrece una interfaz bien definida para la funcionalidad que se invoca.

Consumer En un sistema de mensajes como Kakfa o RabbitMQ, se denomina *consumer* al actor que consume dichos mensajes de la cola.

DTO Un objeto de transferencia de datos, comúnmente conocido como DTO, es un objeto que transporta datos entre procesos.

DevOps Acrónimo inglés de development y operations. Es una práctica de ingeniería de software que tiene como objetivo unificar el desarrollo de software (Dev) y la operación del software (Ops).

Edge-service Componente que está expuesto al Internet público. Actúa como puerta de acceso a todos los demás servicios.

Endpoint Punto de entrada a un servicio, un proceso o un destino de cola o tema en la arquitectura orientada a servicios.

Framework Abstracción en la que, un software que proporciona una funcionalidad genérica, puede cambiar mediante un código adicional escrito por el usuario, proporcionando así un software específico de la aplicación.

Front-end Parte del software que interactúa con los usuarios.

GitHub Plataforma de desarrollo colaborativo para alojar proyectos utilizando el sistema de control de versiones Git. Se utiliza principalmente para la creación de código fuente de programas de computadora.

Open-source Tipo de software en el que el código fuente se publica bajo una licencia en la que el titular de los derechos de autor otorga a los usuarios los derechos para estudiar, cambiar y distribuir el software a cualquier persona y para cualquier propósito.

Producer En un sistema de mensajes como Kakfa o RabbitMQ, se denomina *producer* al actor que produce nuevos eventos y los publica en la cola para que puedan ser consumidos.

Rollback En tecnologías de base de datos, es una operación que devuelve la base de datos a un estado anterior.

SOA Acrónimo de *Service Oriented Architecture* es un estilo de arquitectura de TI que se apoya en la orientación a servicios.

Time-to-market Tiempo que toma desde que un producto se concibe hasta que está disponible para la venta.

Trello Software de administración de proyectos con interfaz web, cliente para iOS y android.

1 Introducción

Antes de empezar, es importante explicar varios de los principales conceptos que van a aparecer en el trabajo.

Las arquitecturas de microservicios son un estilo arquitectónico cuyo objetivo consiste en construir una aplicación a partir de un conjunto de servicios más pequeños comunicados entre ellos. Estos representan las diferentes funcionalidades de negocio y deberían poder ser desplegados de forma independiente al resto [1].

Un sistema monolítico, comúnmente conocido como monolito, se denomina a aquella aplicación cuya base de código está unificada, es compilada a la vez y genera un único artefacto [2].

Las arquitecturas orientadas a servicios, más conocidas como *SOA* y precursoras de los microservicios, son también sistemas distribuidos, aunque se diferencian en la taxonomía, granularidad y coordinación de los servicios [3], entre otras cosas.

1.1 Contexto del proyecto

Los sistemas basados en arquitecturas de microservicios han ganado mucha popularidad estos últimos años. Cada vez son más las empresas que se aventuran a utilizar este estilo arquitectónico para sus proyectos, y es que en la actualidad, uno de cada dos profesionales lo utiliza ya en al menos una de sus aplicaciones y, de los que no, un 73% está considerando empezar a hacerlo [4][5].

Para entender el auge de esta nueva corriente, debemos entender el contexto histórico en el que nos encontramos, así como las alternativas de las que disponemos.

Hoy en día, debido a la feroz competencia que existe entre las empresas de carácter tecnológico, tener la capacidad de responder de forma rápida a las necesidades de los usuarios así como a los cambios del mercado puede marcar la diferencia entre posicionarse como el principal competidor o quedar relegado a las últimas posiciones. El éxito o fracaso de un producto depende, en gran medida, de la estrategia en función del tiempo escogida, y este efecto se acentúa aún más en mercados emergentes y altamente cambiantes como lo es el tecnológico [6]. Por ello, muchas empresas han convertido el denominado *time-to-market* en una de sus prioridades.

A demás, el hecho de ofrecer un servicio a millones de usuarios, como es el caso de las redes sociales, hace imperativo disponer de un sistema con una alta disponibilidad, cuyo tiempo de respuesta sea mínimo y sobre todo, que sea capaz de responder a grandes fluctuaciones en la demanda sin resentirse.

Por todo ello, los sistemas basados en arquitecturas de microservicios se han posicionado como la mejor opción para un gran número de empresas frente al resto de alternativas, como los sistemas monolíticos o los basados en arquitecturas orientadas a servicios [5].

1.2 Formulación del problema

A priori, dada su gran adopción, podría parecer que los microservicios solo ofrecen ventajas respecto al resto de alternativas. Sin embargo, estos también presentan ciertos inconvenientes, muchos de ellos inherentes al hecho de ser una arquitectura distribuida.

Como consecuencia, no todos los proyectos son adecuados para este tipo de arquitecturas y el ejercicio de decidir en cual de ellas basar un sistema puede llegar a resultar bastante complejo.

Una mala decisión durante el diseño de un sistema puede tener un gran impacto en los recursos necesarios para llevar a cabo el desarrollo o mantenimiento del mismo. Por ello, disponer de información acerca de las ventajas e inconvenientes de cada una de las alternativas, así como los casos

en los que resulta una buena opción hacer uso de ellas, puede significar la diferencia entre tomar la decisión correcta y la incorrecta.

2 Definición del alcance

2.1 Objetivo

Este trabajo tiene como objetivo el diseño y desarrollo de una red social, siguiendo dos estilos arquitectónicos diferentes: uno monolítico y otro basado en microservicios para, posteriormente, realizar una medición del rendimiento de cada uno así como identificar sus ventajas e inconvenientes.

Con esto se espera ganar conocimiento en el diseño y desarrollo de sistemas distribuidos así como generar datos acerca del desempeño de ambos sistemas que puedan servir como referencia a la hora de tomar decisiones en proyectos futuros.

2.2 Alcance del proyecto

Como se ha explicado en el apartado 2.1, el proyecto consiste en el desarrollo de una red social, por duplicado, usando dos estilos arquitectónicos diferentes. Por lo tanto, se diseñarán e implementarán ambos sistemas y, una vez terminados, se desplegarán en la nube utilizando *AWS*.

Para facilitar el despliegue y configuración de ambos sistemas así como su posterior escalado, se utilizarán plataformas de contenedores, que se encargarán de alojar la aplicación así como bases de datos y cualquier otro sistema necesario.

Posteriormente se realizarán pruebas de rendimiento para medir su desempeño y se compararán los resultados obtenidos en ambos casos para identificar los puntos fuertes de cada uno de los estilos arquitectónicos utilizados así como sus debilidades.

Con el fin de acotar el proyecto y dada la variedad de alternativas, se desarrollarán únicamente dos sistemas: uno monolítico y otro basado en microservicios, dejando fuera cualquier otro estilo arquitectónico.

Quedan también fuera del alcance del proyecto el diseño e implementación del apartado *front-end,* ya que conllevaría mucho tiempo y supondría un desvío del objetivo principal. A demás, no será necesario para realizar las pruebas de rendimiento, pues estas se llevarán a cabo mediante peticiones directamente al *back-end*.

2.3 Posibles obstáculos

Las dificultades que podrían surgir a lo largo del proyecto son diversas, aunque se pueden reducir a cuatro grupos.

2.3.1 Desarrollo del sistema

El desarrollo del sistema monolítico no debería suponer un problema, pues su funcionamiento es relativamente sencillo. Sin embargo, una arquitectura de microservicios está sujeta a toda una infraestructura que hace que la complejidad del sistema se vea incrementada. Esto supone una dificultad añadida, ya que podrían surgir problemas debidos a la falta de experiencia con las tecnologías necesarias para desarrollarlo.

2.3.2 Puesta en producción

La puesta en producción de ambos sistemas es, con diferencia, el mayor obstáculo a superar durante el desarrollo del proyecto. La falta de conocimientos acerca del funcionamiento de plataformas como *AWS*, sumada a la alta complejidad de los sistemas basado en microservicios, podrían suponer retrasos significativos respecto a la planificación. Por ello, será de extrema importancia anticiparse a dichos problemas con el fin de minimizar su impacto.

2.3.3 Análisis de rendimiento

Otra de las dificultades a tener en cuenta son los problemas que pudieran surgir durante el análisis de rendimiento de ambos sistemas.

Este trabajo tiene dos objetivos principales, el de diseñar y desarrollar los sistemas y el de medir y comparar su desempeño. El hecho de no poder llevar a cabo cualquiera de estos dos puntos en su totalidad haría que la calidad del trabajo se viese reducida.

2.3.4 Necesidad de más recursos

Durante la planificación inicial, se ha realizado una estimación de los recursos necesarios para llevar a cabo el proyecto. El hecho de necesitar más recursos de los planeados o de hacerlo durante más tiempo de lo previsto supondría un coste adicional que incrementaría el presupuesto del proyecto.

3 Estado del arte

Aunque algunas empresas ya habían estado explorando las mismas ideas, fue en mayo del 2011, en un *workshop* de arquitectura de software, cuando se utilizó el término "microservicios" por primera vez para definir un estilo arquitectónico común.

Desde entonces han ganado mucha popularidad y numerosos arquitectos de software han analizado este estilo arquitectónico con el fin de identificar sus ventajas e inconvenientes. A continuación se explicarán las fortalezas y los retos que supone su utilización, así como las bibliotecas más conocidas para el desarrollo de estos.

3.1 Ventajas

Diversidad de tecnologías

Uno de los beneficios de que cada servicio sea independiente es que, siempre y cuando se mantenga el contrato con el resto de servicios, estos pueden estar desarrollados en cualquier tecnología.

Esto es muy positivo, ya que no se está atado a un framework concreto, un lenguaje de programación o una base de datos específica. Por ejemplo, dependiendo de la naturaleza de los datos a almacenar en cada servicio y de el uso que se vaya a hacer de ellos se podría utilizar una base de datos relacional, una orientada a documentos o incluso una orientada a grafos, lo que supondría una mejora significativa en el rendimiento sin afectar al resto del sistema.

Resiliencia

Los fallos en un sistema son algo natural y se debe asumir que en algún momento u otro ocurrirán. En una arquitectura basada en microservicios, donde los servicios son independientes y están aislados el uno del otro, resulta sencillo gestionar los errores de forma que estos no se propaguen y el impacto sobre el resto del sistema sea menor. Si un servicio dejara de funcionar totalmente, el resto del sistema aún seguiría funcionando. Sin embargo, en un sistema monolítico, un fallo similar supondría la caída de todo el sistema.

Este concepto es conocido como *Graceful degradation of services*, y es una de las principales ventajas de las arquitecturas basadas en microservicios.

Escalabilidad

En un sistema monolítico, si una pequeña parte tuviera un problema de rendimiento debido, por ejemplo, a un incremento de peticiones, sería necesario escalar todo el sistema como un conjunto para poder responder a este aumento de la demanda.

En un sistema de microservicios, por el contrario, si un servicio estuviera recibiendo más peticiones de lo habitual, podría escalarse de forma independiente al resto del sistema. Esto supondría un mayor aprovechamiento de recursos y, por consiguiente, una reducción de los costes de mantenimiento.

Facilidad de despliegue

Un pequeño cambio en una aplicación monolítica supondría tener que desplegar el sistema completo, con los riesgos que eso conlleva.

En un sistema de microservicios, en cambio, una modificación únicamente implicaría tener que desplegar el servicio que se ha cambiado. Esto supondría un ahorro en tiempo, pues no habría que ejecutar todos los tests si no solo los del servicio en cuestión, lo que mejoraría de forma significativa el *time-to-market.*

Además, dado que los cambios pueden tener efectos colaterales, el uso de microservicios ayudaría a identificar la fuente de posibles problemas, pues

estos corresponderían a un servicio concreto y sería sencillo realizar un *rollback* de los cambios.

Alineamiento organizacional

Los microservicios ayudan a alinear la arquitectura del sistema con la organización pudiendo asignar servicios a equipos reducidos, en lugar de trabajar todo el mundo en un mismo sistema. Esto ayudaría a incrementar la productividad de los equipos.

También permiten llevar a cabo la metodología ideada por Amazon *"You build it, you run it"*, donde en lugar de centrar todos los esfuerzos en desarrollar el sistema y una vez acabado disolver el equipo, este asume la responsabilidad, tanto del desarrollo como de la puesta producción y de su mantenimiento.

Reusabilidad

El hecho de estar dividido en servicios independientes hace que, en un sistema de microservicios, la reutilización de funcionalidades sea extremadamente sencilla.

Un sistema monolítico, por el contrario, no permite reutilizar funcionalidades de forma tan natural. Un diseño modular, donde se respeten los límites de cada módulo reduciendo el acoplamiento haría esta tarea algo mas sencilla. Sin embargo, evitar el acoplamiento por completo puede ser muy complicado ya que los límites de los módulos no son físicos, como en el caso de los microservicios y tomar "atajos" resulta bastante sencillo.

Cambiabilidad

En un sistema monolítico, un cambio puede afectar a todo el sistema, por lo que si el diseño de este no es bueno o se ha ido deteriorando con el tiempo mediante el uso de malas prácticas, cambiar algo podría conllevar un gran riesgo.

Los sistemas de microservicios no son una excepción, aunque el riesgo que conlleva un cambio es significativamente menor, ya que en el peor de los casos

sólo se vería afectado el servicio que se modifica, y no todo el sistema. Por ejemplo, una estrategia existente para minimizar el impacto de problemas introducidos al realizar cambios son los *"automatic rollouts"* que permiten introducir los cambios de forma gradual en las instancias del servicio mientras se monitorizan los errores [12].

3.2 Retos

Alta modularización

El hecho de dividir un sistema en diferentes servicios presenta muchas ventajas, pero también puede resultar un problema si no se hace adecuadamente. Utilizar pocos servicios supondría no utilizar todo el potencial que los microservicios ofrecen. Utilizar de más implicaría un menor rendimiento debido al coste que conlleva la comunicación entre ellos y un mayor coste en la parte de operaciones para gestionar la infraestructura.

Diversidad de tecnologías

La diversidad de tecnologías, una de las principales ventajas que ofrece el uso de microservicios también puede convertirse en un problema, ya que una empresa podría verse soportando decenas de ellas si no se controla. Esto complicaría el cambio de equipo de programadores, pues las tecnologías podrían ser totalmente diferentes.

Dependencia en DevOps

Debido a la complejidad del un sistema de microservicios, se crea una gran dependencia con roles de *DevOps,* necesarios para mantener la infraestructura y para desplegar y monitorizar los servicios.

Sincronización de datos

Debido a la distribución de los datos, se da por hecho que la consistencia es eventual, por lo que en un momento dado podría haber una falta de sincronización entre bases de datos. Sin embargo, es necesario implementar

un sistema que garantice la consistencia de estos en caso de haber un problema durante una transacción que afecte a dos o mas servicios.

Seguridad

Al usar una arquitectura distribuida, toda la comunicación entre servicios se hace a través de la red. Por ello, garantizar la seguridad de la información enviada gana es de vital importancia en este tipo de sistemas.

Rendimiento

El rendimiento de un sistema puede verse afectado negativamente por el uso de microservicios debido a que la comunicación por red es significativamente mas lenta que a través de la memoria de un ordenador. Para muchos sistemas, esta diferencia sería insignificante, aunque es algo a tener en cuenta a la hora de realizar el diseño [13].

3.3 Herramientas y bibliotecas

En la actualidad existen numerosas bibliotecas que facilitan el desarrollo de sistemas de microservicios. Las más conocidas son aquellas creadas por Netflix, que son *open-source* y se encuentran incluidas en Spring Cloud, uno de los frameworks más utilizados. A continuación se resume el funcionamiento de algunas de esas bibliotecas.

Eureka

Eureka es un servicio encargado de realizar *"service discovery"*. Mantiene un registro de todos los servicios activos y de sus instancias con el fin de llevar un control de los recursos disponibles.

Todos los servicios deberán implementar el cliente de *Eureka*, que se encargará de enviar peticiones cada cierto tiempo al servidor para notificar que estos siguen disponibles. En caso de dejar de recibir mensajes de alguno de los servicios, el servidor asumirá que este ha dejado de funcionar, por lo que lo descartará para que no reciba más peticiones.

Ribbon

Ribbon es un balanceador de carga en cliente que permite balancear las peticiones entre las diferentes instancias de un servicio, que obtiene a través de la integración con *Eureka*.

Por defecto, utiliza un algoritmo *Round Robin* para realizar el balanceo de carga, aunque permite configurar otros algoritmos o incluso definir unos totalmente personalizados.

Zuul

Zuul es un *edge-service* que se encarga de realizar enrutado dinámico, de la monitorización y de la seguridad, entre otros.

Sirve como puerta de entrada al sistema y permite redirigir las peticiones mediante la integración con *Eureka* al servicio pertinente. También se integra con *Ribbon* para poder balancear la carga entre las diferentes instancias disponibles.

Hystrix

Hystrix es una biblioteca que implementa el patrón *circuit breaker*. Esta ayuda a minimizar el impacto de posibles errores ya que al detectar que una instancia no está funcionando correctamente, ya sea por un error o porque su tiempo de respuesta es demasiado elevado, permite ejecutar una operación alternativa y hacer que dicha instancia quede inaccesible hasta que se recupere.

Archaius

Archaius es una biblioteca de configuración en cliente que permite mantener la configuración de los clientes centralizada. Es una extensión del proyecto *Apache Commons Configuration*.

4 Metodología

4.1 Metodología de trabajo

En un primer momento se pensó en hacer uso de metodologías ágiles, con iteraciones cortas y un *backlog* de tareas que se iría refinando a medida que avanzase el proyecto. Sin embargo, debido a la naturaleza de este, donde el plazo de tiempo es relativamente corto y las todas las especificaciones del sistema deberían definirse de forma previa al desarrollo con el fin de poder realizar una planificación realista, se considera que la metodología en cascada encaja mejor con las necesidades del proyecto.

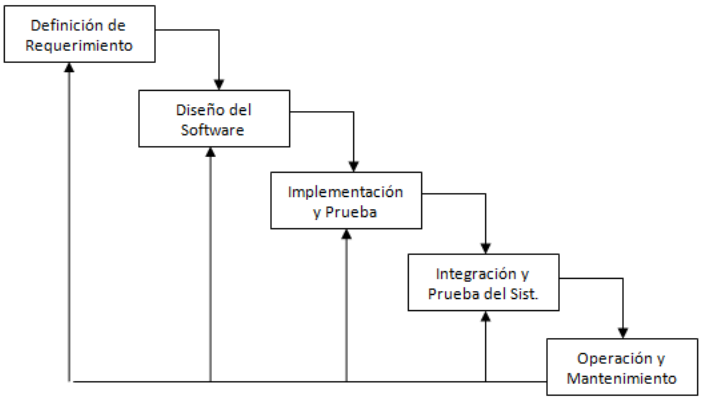


Ilustración 1: Metodología en cascada

Debido a que este trabajo requerirá de dos sistemas que se tratarán como proyectos distintos, cada uno de ellos tendrá su propio ciclo.

Como se puede observar en la Ilustración 1, el primer paso será definir todos los requerimientos del sistema, excepto en el caso de los microservicios, ya que serán los mismos que los del sistema monolítico y por lo tanto podrán ser reaprovechados.

El siguiente paso será realizar el diseño del sistema. Esto incluye la definición de los casos de uso, diagramas de clases y todos aquellos artefactos

necesarios para llevar a cabo el desarrollo. Una vez terminado el diseño se desarrollará el sistema, se integrarán todos los componentes y se probará su correcto funcionamiento.

Para finalizar, se pondrá en producción y entrará en fase de mantenimiento. A partir de este momento no se añadirán nuevas funcionalidades, aunque se arreglarán aquellos problemas identificados que pudieran afectar negativamente a los resultados de las pruebas.

4.2 Herramientas de seguimiento

Para hacer un seguimiento del trabajo realizado durante el desarrollo de cada uno de los sistemas se utilizará *Trello,* una herramienta de gestión de online.

Debido a que el equipo tendrá un único integrante, el uso de estas herramientas no será necesario, ya que no habrá que coordinarse con nadie. Aún así, se considera que podría resultar útil ya que ayudará mejorar la organización de las tareas y a visualizar tanto trabajo realizado como el que queda por hacer.

También se utilizará *GitHub* como repositorio de código, pudiendo llevar así un registro de todo el trabajo realizado además de volver a una versión anterior en caso de ser necesario.

4.3 Método de validación

Para realizar una validación del trabajo realizado se aprovecharán las reuniones con el director. Estas servirán para verificar que el progreso del trabajo concuerda con el especificado en la planificación temporal, así como que los objetivos del proyecto siguen siendo viables.

Para la validación de la parte técnica, se realizarán las pruebas necesarias de ambos sistemas para verificar que el funcionamiento es el esperado.

5 Análisis de requisitos

Una vez definidos el objetivo del proyecto y el alcance del mismo, será necesario especificar los requisitos que el sistema debe cumplir. Para ello, a continuación se identifican los actores implicados, y se listan todos los requisitos funcionales del sistema mediante la definición de casos de uso [16].

5.1 Visión del proyecto

El proyecto a desarrollar consiste en una red social, similar a LinkedIn, que permitirá a los usuarios encontrar trabajo gracias a un sistema de candidaturas mediante el cual podrán presentarse candidatos a ofertas que empresas hayan publicado en el sistema.

Además, los usuarios también podrán introducir en el sistema sus experiencias, tanto académicas como profesionales, así como interactuar con otros usuarios ya sea siguiéndoles para estar al día del contenido que estos publican o mediante el sistema mediante publicaciones.

5.2 Actores implicados

Con el objetivo de estimar el impacto, tanto del desarrollo del proyecto como de los resultados obtenidos a partir de este, es necesario identificar todas y cada una de las partes interesadas, conocidas como *stakeholders*.

En este trabajo, se pueden distinguir hasta cuatro clases de actores implicados: el desarrollador del proyecto, el director, la empresa para la cual trabajo y los beneficiarios.

Desarrollador

Yo, como desarrollador del proyecto, me encargaré de que los objetivos definidos en la sección 2.1 se cumplan al finalizar el trabajo así como de desarrollar y documentar todo el proceso.

También seré beneficiario directo, pues ganaré conocimientos en un tema de especial interés para aplicarlos en proyectos futuros.

Director

El director de proyecto, Ernest Teniente, quien se encargará de supervisar el trabajo realizado, ayudando si fuese necesario y velando por que se cumplan todos los objetivos establecidos.

Empresa

La empresa donde trabajo actualmente se encuentra en mitad de una migración de un sistema monolítico a uno basado en microservicios. Por ello, esta se beneficiará directamente de los conocimientos que adquiera, tanto del funcionamiento de sistemas basados en microservicios como de herramientas y tecnologías necesarias para su desarrollo.

Beneficiarios

Como beneficiarios se incluyen todas aquellas personas interesadas en arquitectura de software que, una vez acabado el proyecto, hagan uso total o parcial del código desarrollado así como de los resultados obtenidos.

5.3 Casos de uso

A continuación se definen los requisitos del sistema en forma de casos de uso, que son la especificación del conjunto de acciones que realiza un sistema y que permiten obtener un resultado útil para uno o más actores [16].

Dado que el sistema a desarrollar ofrece servicios web y no dispone de un *front-end* para que los usuarios lo utilicen directamente, el único actor será

un sistema de software externo, al que de ahora en adelante se referirá como cliente web.

5.3.1 Diagrama de casos de uso

Con el fin de simplificar el diagrama de casos de uso, estos se han separado por las siguientes áreas funcionales: usuarios, ofertas de empleo, publicaciones y experiencias.

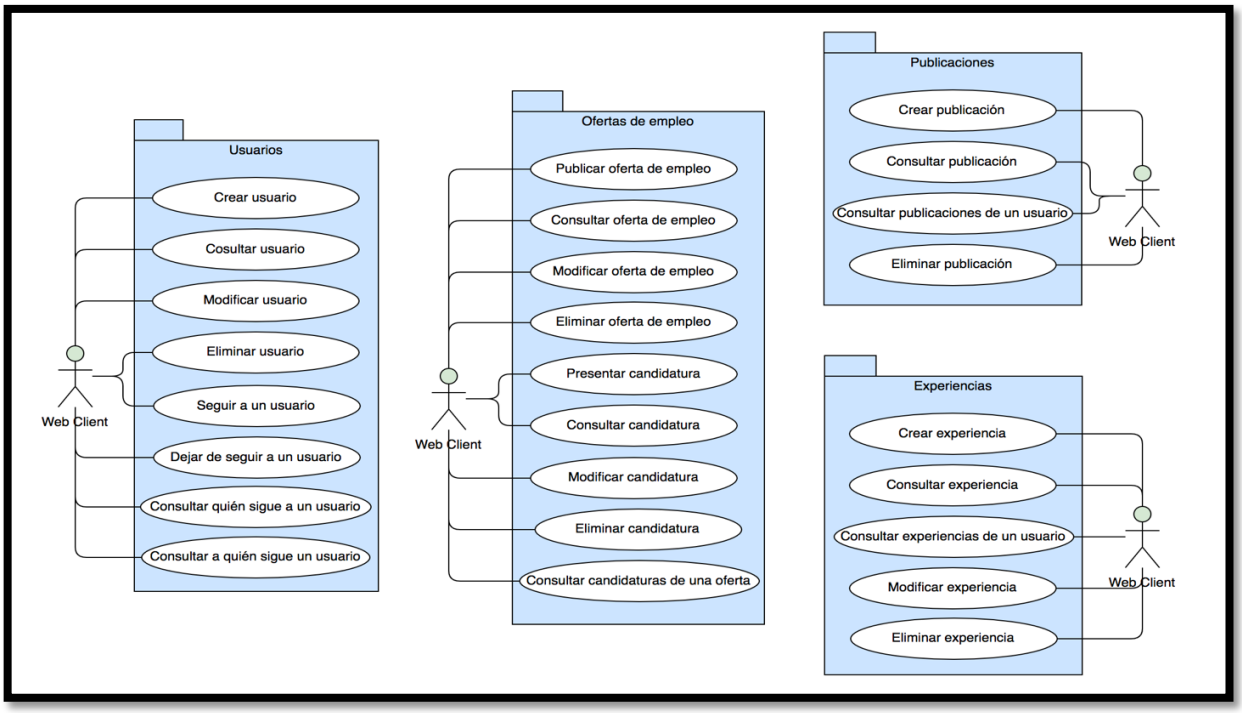


Ilustración 2: Diagrama de casos de uso

5.3.2 Especificación brief style

Usuarios

- **CU001 Crear usuario:** El cliente podrá crear un nuevo usuario en el sistema, que podrá ser particular o empresa.
- **CU002 Consultar usuario:** El cliente podrá consultar la información referente a cualquier usuario existente en el sistema.

- **CU003 Modificar usuario:** El cliente podrá modificar la información de cualquier usuario existente en el sistema.
- **CU004 Eliminar usuario:** El cliente podrá eliminar cualquier usuario existente en el sistema.
- **CU005 Seguir a usuario:** El cliente podrá crear un nuevo seguimiento de un usuario del sistema a otro distinto.
- **CU006 Dejar de seguir a usuario:** El cliente podrá hacer que un usuario deje de seguir a otro.
- **CU007 Consultar quien sigue a un usuario:** El cliente podrá ver qué usuarios del sistema siguen a un usuario concreto.
- **CU008 Consultar a quien sigue un usuario:** El cliente podrá consultar a que usuarios del sistema sigue un usuario concreto.

Ofertas de empleo

- **CU009 Publicar oferta de empleo:** El cliente podrá crear una nueva oferta de empleo vinculada a un usuario de tipo empresa existente en el sistema.
- **CU010 Consultar oferta de empleo:** El cliente podrá consultar toda la información referente a una oferta de empleo existente en el sistema.
- **CU011 Modificar oferta de empleo:** El cliente podrá modificar los datos de una oferta de empleo existente en el sistema.
- **CU012 Eliminar oferta de empleo:** El cliente podrá eliminar una oferta de empleo existente en el sistema.
- **CU013 Presentar candidatura:** El cliente podrá crear una candidatura vinculada a un usuario y a una oferta de empleo existentes en el sistema.
- **CU014 Consultar candidatura:** El cliente podrá consultar la información referente a la candidatura de un usuario para una oferta de empleo.
- **CU015 Modificar candidatura:** El cliente podrá modificar la información de una candidatura existente en el sistema.
- **CU016 Eliminar candidatura:** El cliente podrá eliminar cualquier candidatura existente en el sistema.

- **CU017 Consultar candidaturas de una oferta:** El cliente podrá consultar las candidaturas que se han presentado para una oferta de empleo concreta.

Publicaciones

- **CU018 Crear publicación:** El cliente podrá crear una publicación vinculada a un usuario.
- **CU019 Consultar publicación:** El cliente podrá consultar la información referente a una publicación.
- **CU020 Consultar publicaciones de un usuario:** El cliente podrá consultar las publicaciones que ha realizado un usuario concreto.
- **CU021 Eliminar publicación:** El cliente podrá eliminar cualquier publicación del sistema.

Experiencias

- **CU022 Añadir experiencia:** El cliente podrá crear una experiencia vinculada a un usuario, que podrá ser profesional o académica.
- **CU023 Consultar experiencia:** El cliente podrá consultar toda la información referente a un experiencia.
- **CU024 Consultar experiencias de un usuario:** El cliente podrá consultar todas las experiencias vinculadas a un usuario concreto.
- **CU025 Modificar experiencia:** El cliente podrá modificar experiencias existentes en el sistema.
- **CU026 Eliminar experiencia:** El cliente podrá eliminar cualquier experiencia existente en el sistema.

5.3.3 Especificación completa

Caso de uso	**CU001 Crear usuario**	Actor principal	**Cliente**
Disparador	El cliente realiza una petición para crear un usuario.		
Escenario principal de éxito			
1. El cliente hace una petición al sistema para crear un usuario. 2. El sistema crea un nuevo usuario con los datos proporcionados. 3. El sistema retorna el usuario creado.			
Extensiones			
2.1 Los datos proporcionados no son válidos. 2.1.1 El sistema retorna un error al cliente			

Caso de uso	**CU002 Consultar usuario**	Actor principal	**Cliente**
Disparador	El cliente realiza una petición para consultar los datos de un usuario.		
Escenario principal de éxito			
1. El cliente hace una petición al sistema para consultar los datos de un usuario. 2. El sistema obtiene los datos del usuario. 3. El sistema retorna los datos del usuario.			
Extensiones			
2.1 El usuario especificado no existe en el sistema. 2.1.1 El sistema retorna un error al cliente.			

Caso de uso	**CU003 Modificar usuario**	Actor principal	**Cliente**
Disparador	El cliente realiza una petición para modificar un usuario.		
Escenario principal de éxito			
1. El cliente hace una petición al sistema para modificar un usuario. 2. El sistema modifica el usuario con los nuevos datos proporcionados. 3. El sistema retorna el usuario modificado.			
Extensiones			
2.1 El usuario especificado no existe en el sistema. 2.1.1 El sistema retorna un error al cliente. 2.2 Los datos proporcionados no son válidos. 2.2.1 El sistema retorna un error al cliente			

Caso de uso	**CU004 Eliminar usuario**	Actor principal	**Cliente**
Disparador	El cliente realiza una petición para eliminar un usuario.		
Escenario principal de éxito			
1. El cliente hace una petición al sistema para eliminar un usuario. 2. El sistema elimina el usuario así como todas sus candidaturas, publicaciones, experiencias y *follows*.			
Extensiones			
2.1 El usuario especificado no existe en el sistema. 2.1.1 El sistema retorna un error al cliente.			

Caso de uso	**CU005 Seguir a usuario**	Actor principal	**Cliente**
Disparador	El cliente realiza una petición para realizar que un usuario siga a otro.		
Escenario principal de éxito			
1. El cliente realiza una petición para realizar que un usuario siga a otro. 2. El sistema crea el *follow* entre dos usuarios.			
Extensiones			
2.1 Alguno de los usuarios especificados no existe en el sistema. 2.1.1 El sistema retorna un error al cliente. 2.2 El usuario especificado ya sigue al otro. 2.2.1 El sistema retorna un error al cliente.			

Caso de uso	**CU006 Dejar de seguir a usuario**	Actor principal	**Cliente**
Disparador	El cliente realiza una petición para que un usuario deje de seguir a otro.		
Escenario principal de éxito			
1. El cliente realiza una petición para que un usuario deje de seguir a otro. 2. El sistema elimina el *follow* entre los dos usuarios.			
Extensiones			
2.1 Alguno de los usuarios especificados no existe en el sistema. 2.1.1 El sistema retorna un error al cliente. 2.2 El *follow* especificado no existe en el sistema. 2.2.1 El sistema retorna un error al cliente.			

Caso de uso	**CU007 Consultar quien sigue a un usuario**	Actor principal	**Cliente**
Disparador	El cliente realiza una petición para consultar quien sigue a un usuario.		
Escenario principal de éxito			
1. El cliente hace una petición al sistema para consultar quien sigue a un usuario. 2. El sistema obtiene la lista de usuarios que siguen al usuario especificado. 3. El sistema retorna la lista.			
Extensiones			
2.1 El usuario especificado no existe en el sistema. 2.1.1 El sistema retorna un error al cliente.			

Caso de uso	**CU008 Consultar a quien sigue un usuario**	Actor principal	**Cliente**
Disparador	El cliente realiza una petición para consultar a quien sigue un usuario.		
Escenario principal de éxito			
1. El cliente realiza una petición para consultar a quien sigue un usuario. 2. El sistema obtiene la lista de usuarios a los que sigue el usuario especificado. 3. El sistema retorna la lista.			
Extensiones			
2.1 El usuario especificado no existe en el sistema. 2.1.1 El sistema retorna un error al cliente.			

Caso de uso	**CU009 Publicar oferta de empleo**	Actor principal	**Cliente**
Disparador	El cliente realiza una petición para crear una oferta de empleo.		
Escenario principal de éxito			
1. El cliente realiza una petición para crear una oferta de empleo. 2. El sistema crea la oferta de empleo en el sistema, vinculada a la empresa especificada. 3. El sistema retorna la nueva oferta de empleo.			
Extensiones			
2.1 La empresa especificada no existe en el sistema. 2.1.1 El sistema retorna un error al cliente. 2.2 Los datos proporcionados no son válidos. 2.2.1 El sistema retorna un error al cliente			

Caso de uso	**CU010 Consultar oferta de empleo**	Actor principal	**Cliente**
Disparador	El cliente realiza una petición para consultar una oferta de empleo.		
Escenario principal de éxito			
1. El cliente realiza una petición para consultar una oferta de empleo. 2. El sistema obtiene los datos de la oferta de empleo especificada. 3. El sistema retorna la oferta.			
Extensiones			
2.1 La oferta de empleo especificada no existe en el sistema. 2.1.1 El sistema retorna un error al cliente.			

Caso de uso	**CU011 Modificar oferta de empleo**	Actor principal	**Cliente**
Disparador	El cliente realiza una petición para modificar una oferta de empleo.		
Escenario principal de éxito			
1. El cliente realiza una petición para modificar una oferta de empleo. 2. El sistema modifica la oferta de empleo especificada. 3. El sistema retorna la nueva oferta de empleo.			
Extensiones			
2.1 La oferta de empleo especificada no existe en el sistema. 2.1.1 El sistema retorna un error al cliente. 2.2 Los datos proporcionados no son válidos. 2.2.1 El sistema retorna un error al cliente			

Caso de uso	**CU012 Eliminar oferta de empleo**	Actor principal	**Cliente**
Disparador	El cliente realiza una petición para eliminar una oferta de empleo.		
Escenario principal de éxito			
1. El cliente hace una petición al sistema para eliminar una oferta de empleo. 2. El sistema elimina la oferta de empleo.			
Extensiones			
2.1 La oferta de empleo especificada no existe en el sistema. 2.1.1 El sistema retorna un error al cliente.			

Caso de uso	**CU013 Presentar candidatura**	Actor principal	**Cliente**
Disparador	El cliente realiza una petición para presentar una candidatura.		
Escenario principal de éxito			
1. El cliente hace una petición al sistema para presentar una candidatura. 2. El sistema crea una candidatura vinculada a un usuario y una oferta de empleo. 3. El sistema retorna la nueva candidatura.			
Extensiones			
2.1 El usuario especificado no existe en el sistema. 2.1.1 El sistema retorna un error al cliente. 2.2 La oferta especificada no existe en el sistema. 2.2.1 El sistema retorna un error al cliente. 2.3 Ya existe una candidatura del usuario hacia la oferta especificada. 2.3.1 El sistema retorna un error al cliente. 2.4 Los datos proporcionados no son válidos. 2.4.1 El sistema retorna un error al cliente			

Caso de uso	**CU014 Consultar candidatura**	Actor principal	**Cliente**
Disparador	El cliente realiza una petición para consultar una candidatura.		
Escenario principal de éxito			
1. El cliente hace una petición al sistema consultar una candidatura. 2. El sistema obtiene los datos de la candidatura especificada. 3. El sistema retorna la candidatura.			
Extensiones			
2.1 La candidatura especificada no existe en el sistema. 2.1.1 El sistema retorna un error al cliente.			

Caso de uso	**CU015 Modificar candidatura**	Actor principal	**Cliente**
Disparador	El cliente realiza una petición para modificar una candidatura.		
Escenario principal de éxito			
1. El cliente hace una petición al sistema para modificar una candidatura. 2. El sistema modifica la candidatura especificada con los datos proporcionados. 3. El sistema retorna la candidatura modificada.			
Extensiones			
2.1 La candidatura especificada no existe en el sistema. 2.1.1 El sistema retorna un error al cliente. 2.2 Los datos proporcionados no son válidos. 2.2.1 El sistema retorna un error al cliente			

Caso de uso	**CU016 Eliminar candidatura**	Actor principal	**Cliente**
Disparador	El cliente realiza una petición para eliminar una candidatura.		
Escenario principal de éxito			
1. El cliente hace una petición al sistema para eliminar una candidatura. 2. El sistema elimina la candidatura.			
Extensiones			
2.1 La candidatura especificada no existe en el sistema. 2.1.1 El sistema retorna un error al cliente.			

Caso de uso	**CU017 Consultar candidaturas de una oferta**	Actor principal	**Cliente**
Disparador	El cliente realiza una petición para consultar las candidaturas de una oferta.		
Escenario principal de éxito			
1. El cliente hace una petición al sistema para consultar las candidaturas de una oferta. 2. El sistema obtiene todas las candidaturas vinculadas a la oferta especificada 3. El sistema retorna la lista de candidaturas.			
Extensiones			
2.1 La oferta especificada no existe en el sistema. 2.1.1 El sistema retorna un error al cliente.			

Caso de uso	**CU018 Crear publicación**	Actor principal	**Cliente**
Disparador	El cliente realiza una petición para crear una publicación.		
Escenario principal de éxito			
1. El cliente hace una petición al sistema para crear una publicación. 2. El sistema crea una nueva publicación vinculada al usuario especificado. 3. El sistema retorna la publicación.			
Extensiones			
2.1 El usuario especificado no existe en el sistema. 2.1.1 El sistema retorna un error al cliente. 2.2 Los datos proporcionados no son válidos. 2.2.1 El sistema retorna un error al cliente			

Caso de uso	**CU019 Consultar publicación**	Actor principal	**Cliente**
Disparador	El cliente realiza una petición para consultar una publicación.		
Escenario principal de éxito			
1. El cliente hace una petición al sistema para consultar una publicación. 2. El sistema obtiene los datos de la publicación especificada. 3. EL sistema retorna la publicación.			
Extensiones			
2.1 La publicación especificada no existe en el sistema. 2.1.1 El sistema retorna un error al cliente.			

Caso de uso	**CU020 Consultar publicaciones de un usuario**	Actor principal	**Cliente**
Disparador	El cliente realiza una petición para consultar las publicaciones de un usuario.		
Escenario principal de éxito			
1. El cliente hace una petición al sistema para consultar las publicaciones de un usuario. 2. El sistema obtiene todas las publicaciones del usuario especificado. 3. El sistema retorna la lista de publicaciones.			
Extensiones			
2.1 El usuario especificado no existe en el sistema. 2.1.1 El sistema retorna un error al cliente.			

Caso de uso	**CU021 Eliminar publicación**	Actor principal	**Cliente**
Disparador	El cliente realiza una petición para eliminar una publicación.		
Escenario principal de éxito			
1. El cliente hace una petición al sistema para eliminar una publicación. 2. El sistema elimina la publicación especificada.			
Extensiones			
2.1 La publicación especificada no existe en el sistema. 2.1.1 El sistema retorna un error al cliente.			

Caso de uso	**CU022 Añadir experiencia**	Actor principal	**Cliente**
Disparador	El cliente realiza una petición para añadir una experiencia.		
Escenario principal de éxito			
1. El cliente hace una petición al sistema para añadir una experiencia. 2. El sistema crea una nueva experiencia vinculada al usuario especificado. 3. El sistema retorna la nueva experiencia.			
Extensiones			
2.1 El usuario especificado no existe en el sistema. 2.1.1 El sistema retorna un error al cliente. 2.2 Los datos proporcionados no son válidos. 2.2.1 El sistema retorna un error al cliente			

Caso de uso	**CU023 Consultar experiencia**	Actor principal	**Cliente**
Disparador	El cliente realiza una petición para consultar una experiencia.		
Escenario principal de éxito			
1. El cliente hace una petición al sistema para consultar una experiencia. 2. El sistema obtiene los datos de la experiencia especificada. 3. El sistema retorna la experiencia.			
Extensiones			
2.1 La experiencia especificada no existe en el sistema. 2.1.1 El sistema retorna un error al cliente.			

Caso de uso	**CU024 Consultar experiencias de un usuario**	Actor principal	**Cliente**
Disparador	El cliente realiza una petición para consultar las experiencias de un usuario.		
Escenario principal de éxito			
1. El cliente hace una petición al sistema para consultar las experiencias de un usuario. 2. El sistema obtiene todas las experiencias vinculadas al usuario especificado. 3. El sistema retorna la lista de experiencias.			
Extensiones			
2.1 El usuario especificado no existe en el sistema. 2.1.1 El sistema retorna un error al cliente.			

Caso de uso	**CU025 Modificar experiencia**	Actor principal	**Cliente**
Disparador	El cliente realiza una petición para modificar una experiencia.		
Escenario principal de éxito			
1. El cliente hace una petición al sistema para modificar una experiencia. 2. El sistema modifica la experiencia especificada con los datos proporcionados. 3. El sistema retorna la experiencia modificada.			
Extensiones			
2.1 La experiencia especificada no existe en el sistema. 2.1.1 El sistema retorna un error al cliente. 2.2 Los datos proporcionados no son válidos. 2.2.1 El sistema retorna un error al cliente			

Caso de uso	**CU026 Eliminar experiencia**	Actor principal	**Cliente**
Disparador	El cliente realiza una petición para eliminar una experiencia.		
Escenario principal de éxito			
1. El cliente hace una petición al sistema para eliminar una experiencia. 2. El sistema elimina la experiencia.			
Extensiones			
2.1 La experiencia especificada no existe en el sistema. 2.1.1 El sistema retorna un error al cliente.			

6 Especificación

6.1 Esquema conceptual

6.1.1 Descripción de las clases

Usuario

La clase *User* representa a una persona física que dispone de una cuenta en el sistema. Esta es una subclase de *AbstractUser*, que implementa campos como id, nombre o la fecha de creación, que son comunes con otra clase. Además de dichos campos, esta contiene otra información relevante como el apellido del usuario y su fecha de nacimiento.

Empresa

La clase *Company* representa una empresa que dispone de una cuenta en el sistema. Al igual que *User*, esta también es subclase de *AbstractUser*, por lo que también dispone de los campos id, nombre y fecha de creación. Además de estos, también incluye el campo descripción.

Publicación

La clase *Post* representa una publicación realizada por uno de los usuarios del sistema. Esta incluye el texto publicado y la fecha de publicación, además de un id y del id del usuario que la realizó.

Experiencia laboral

La clase *EducationExperience* representa una experiencia académica de un usuario del sistema. Esta es una subclase de *Experience*, que implementa campos comunes para el resto de tipos de experiencia como un id, el id del usuario al cual pertenece, una descripción y fechas de inicio y fin. Además de dichos campos, *EducationExperience* también incluye otra información

relevante como el centro donde tuvo lugar, el nombre de los estudios cursados, el campo y la nota.

Experiencia profesional

La clase *WorkExperience* representa una experiencia laboral perteneciente a un usuario del sistema. Al igual que *AcademicExperience*, esta también es una subclase de *Experience,* por lo que también contiene un id, el id del usuario a la cual pertenece, una descripción y fechas de inicio y fin. Además de estos campos comunes, *WorkExperience* incluye otra información relevante como el puesto, la empresa, el lugar donde se realizó y el sector.

Oferta de empleo

La clase *JobOffer* representa una oferta de empleo que ha sido publicada por una empresa. Esta contiene información como la empresa que la publica, el título de la oferta, una descripción, el tipo de contrato, el sector, la localización, el salario, las funciones a realizar y las habilidades requeridas. También incluye las fechas de creación y modificación de la oferta.

Candidatura

La clase *Application* representa la candidatura que un usuario presenta para una oferta. Esta incluye los identificadores de la empresa que publicó la oferta así como del usuario candidato, además de la fecha en la que se realizó y una carta de presentación.

Tipo contrato

La enumeración *EmploymentType* incluye un listado de tipos de contrato que podrán ser utilizados al crear una oferta. Un ejemplo serían contrato fijo y contrato de prácticas.

6.1.2 Diagrama de clases

A continuación se muestra el diagrama de clases, que incluye todas las clases mencionadas en el apartado anterior junto con sus relaciones.

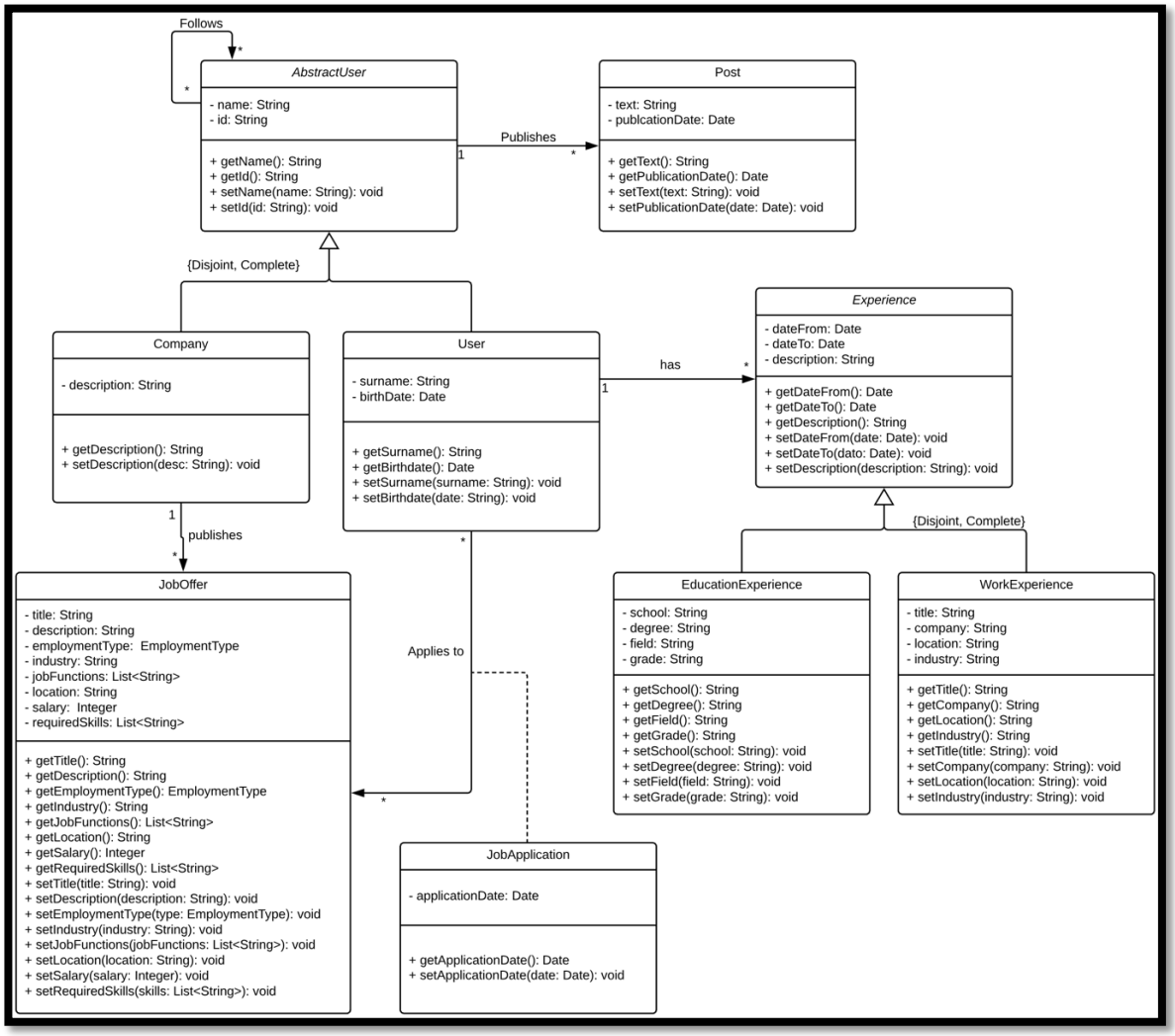


Ilustración 3: Esquema conceptual del sistema

6.2 Esquema de comportamiento

Una vez realizado el esquema conceptual y con el objetivo de definir como debería ser la interacción del cliente web con el sistema así como los contratos de la operaciones, se ha procedido a realizar un diagrama de secuencia para cada uno de los casos de uso especificados en la sección 5.3.

Caso de uso CU001

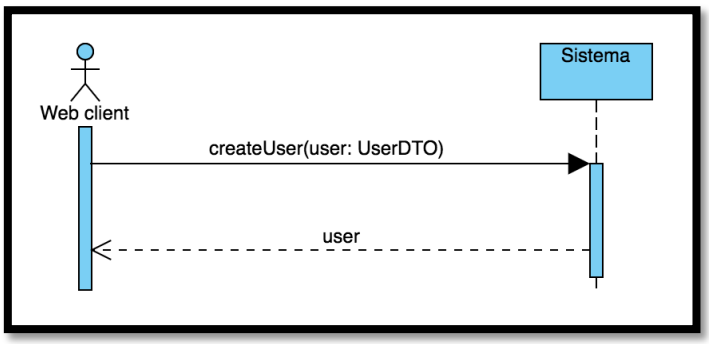


Ilustración 4: Diagrama de secuencia del caso de uso UC001

Contexto	Sistema::createUser(user: UserDTO)
Precondición	El usuario proporcionado es válido.
Postcondición	Se retorna el usuario *user*, que ha sido creado en el sistema.

Caso de uso CU002

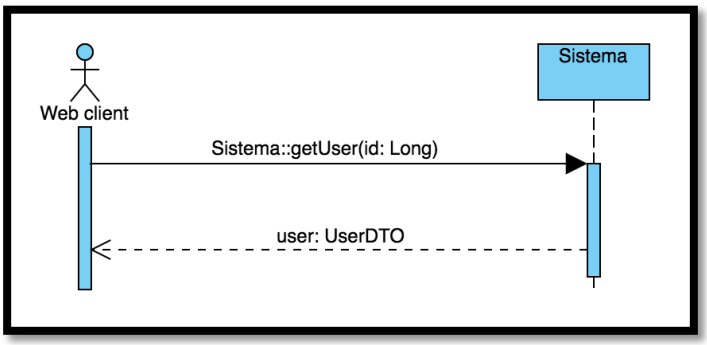


Ilustración 5: Diagrama de secuencia del caso de uso UC002

Contexto	Sistema::getUser(id: Long)
Precondición	El usuario existe en el sistema.
Postcondición	Se retorna el usuario con el id especificado.

Caso de uso CU003

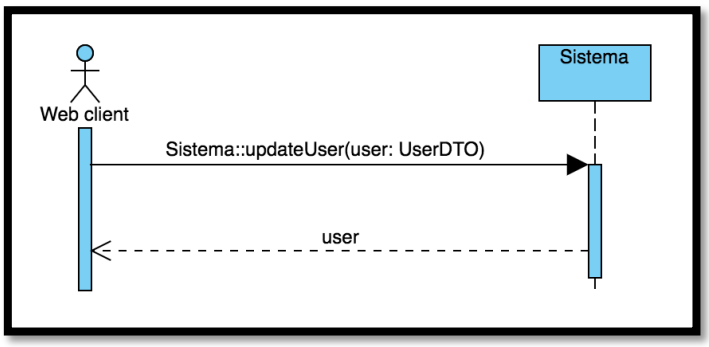


Ilustración 6: Diagrama de secuencia del caso de uso UC003

Contexto	Sistema::updateUser(user: UserDTO)
Precondición	El usuario proporcionado es válido y ya existe.
Postcondición	Se retorna el usuario *user*, que ha sido modificado.

Caso de uso CU004

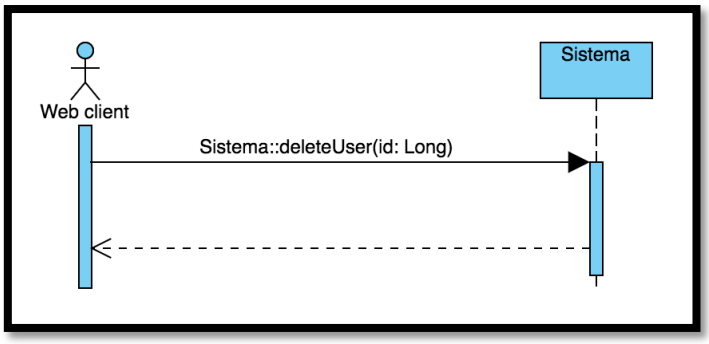


Ilustración 7: Diagrama de secuencia del caso de uso UC004

Contexto	Sistema::deleteUser(id: Long)
Precondición	El usuario existe en el sistema.
Postcondición	El usuario con id especificado ha sido eliminado.

Caso de uso CU005

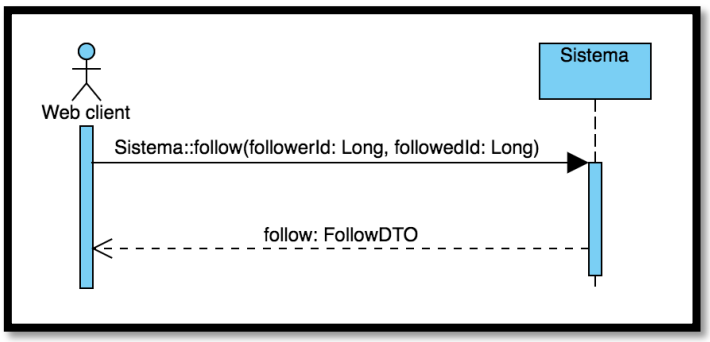


Ilustración 8: Diagrama de secuencia del caso de uso UC005

Contexto	Sistema::follow(followerId: Long, followedId: Long)
Precondición	Los dos usuarios existen y no se siguen.
Postcondición	El usuario con id *followerId* ha comenzado a seguir al usuario con id *followedId.*

Caso de uso CU006

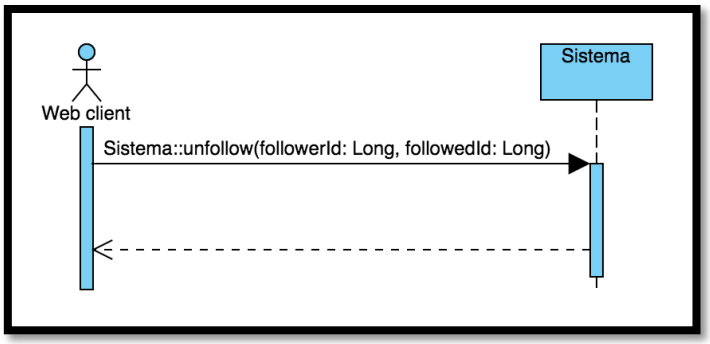


Ilustración 9: Diagrama de secuencia del caso de uso UC006

Contexto	Sistema::unfollow(followerId: Long, followedId: Long)
Precondición	Los dos usuarios existen y se siguen.
Postcondición	El usuario con id *followerId* ha dejado de seguir al usuario con id *followedId.*

Caso de uso CU007

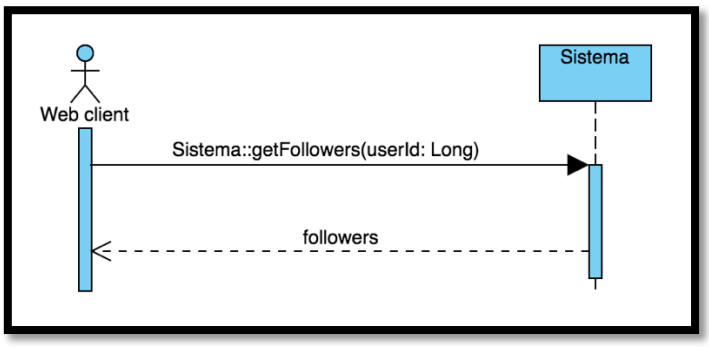


Ilustración 10: Diagrama de secuencia del caso de uso UC007

Contexto	Sistema::getFollowers(userId: Long)
Precondición	El usuario con id *userId* existe.
Postcondición	Se retorna la lista de usuarios que siguen al usuario con id *userId*.

Caso de uso CU008

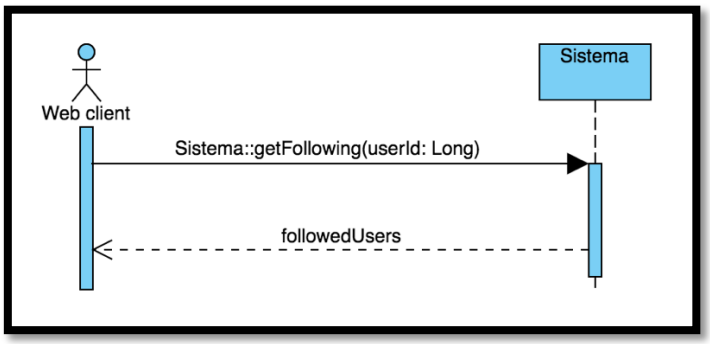


Ilustración 11: Diagrama de secuencia del caso de uso UC008

Contexto	Sistema::getFollowing(userId: Long)
Precondición	El usuario con id *userId* existe.
Postcondición	Se retorna la lista de usuarios a los que sigue el usuario con id *userId*.

Caso de uso CU009

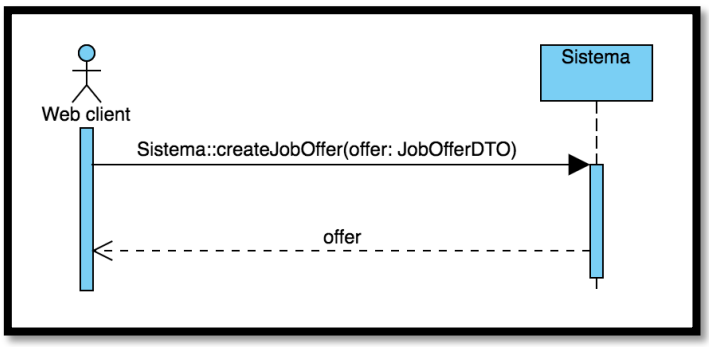


Ilustración 12: Diagrama de secuencia del caso de uso UC009

Contexto	Sistema::createJobOffer(offer: JobOfferDTO)
Precondición	La oferta proporcionada es válida.
Postcondición	Se retorna el la oferta *offer*, que ha sido ha creada en el sistema.

Caso de uso CU010

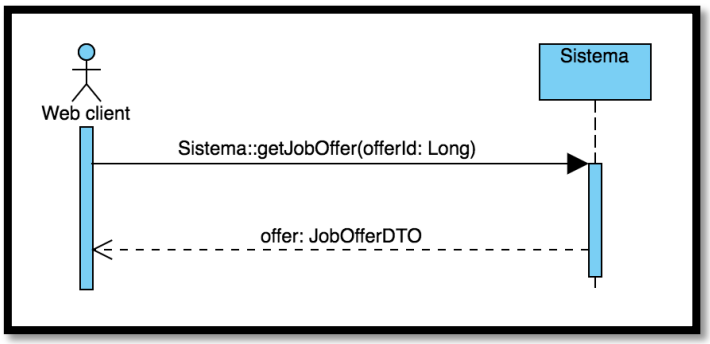


Ilustración 13: Diagrama de secuencia del caso de uso UC010

Contexto	Sistema::getJobOffer(offerId: Long)
Precondición	El oferta con id *offerId* existe.
Postcondición	Se retorna la oferta con id *offerId.*

Caso de uso CU011

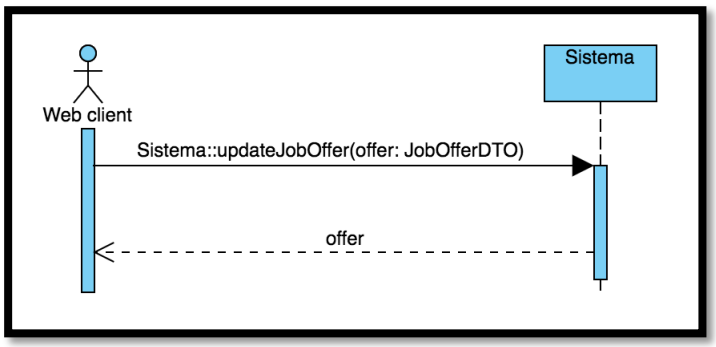


Ilustración 14: Diagrama de secuencia del caso de uso UC011

Contexto	Sistema::updateJobOffer(offer: JobOfferDTO)
Precondición	La oferta proporcionada existe y es válida.
Postcondición	Se retorna la oferta *offer*, que ha sido modificada.

Caso de uso CU012

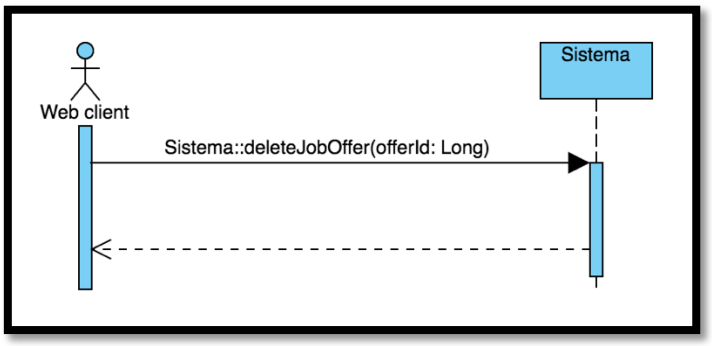


Ilustración 15: Diagrama de secuencia del caso de uso UC012

Contexto	Sistema::deleteJobOffer(offerId: Long)
Precondición	La oferta con id offerId existe.
Postcondición	La oferta con id offerId ha sido eliminada.

Caso de uso CU013

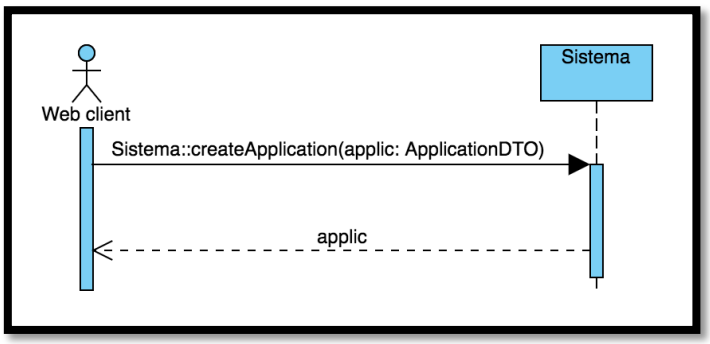


Ilustración 16: Diagrama de secuencia del caso de uso UC013

Contexto	Sistema::createApplication(applic: ApplicationDTO)
Precondición	La candidatura proporcionada es válida.
Postcondición	Se retorna la candidatura *applic*, que ha sido ha creada en el sistema.

Caso de uso CU014

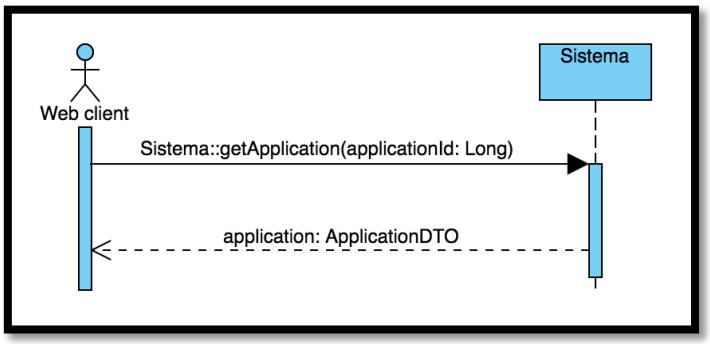


Ilustración 17: Diagrama de secuencia del caso de uso UC014

Contexto	Sistema::getApplication(applicationId: Long)
Precondición	La candidatura con id *applicationId* existe.
Postcondición	Se retorna el la candidatura con id *applicationId*.

Caso de uso CU015

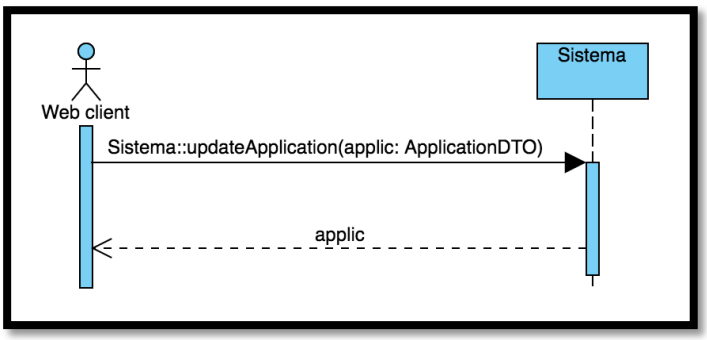


Ilustración 18: Diagrama de secuencia del caso de uso UC015

Contexto	Sistema::updateApplication(applic: ApplicationDTO)
Precondición	La candidatura proporcionada existe y es válida.
Postcondición	Se retorna el la candidatura *applic*, que ha sido ha modificada.

Caso de uso CU016

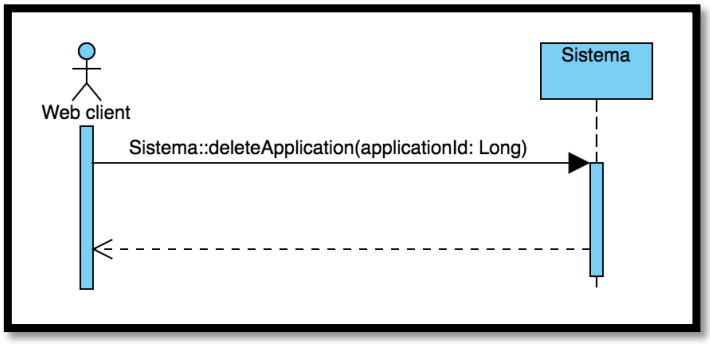


Ilustración 19: Diagrama de secuencia del caso de uso UC016

Contexto	Sistema::deleteApplication(applicationId: Long)
Precondición	La candidatura con id *applicatinId* existe.
Postcondición	La candidatura con id *applicatinId* ha sido eliminada.

Caso de uso CU017

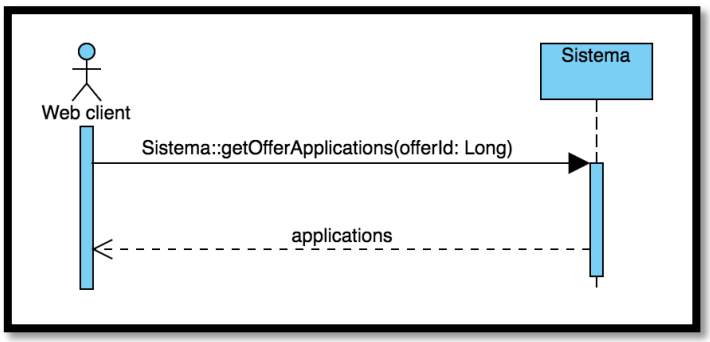


Ilustración 20: Diagrama de secuencia del caso de uso UC017

Contexto	Sistema::getOfferApplications(offerId: Long)
Precondición	La oferta con id *offerId* existe.
Postcondición	Se retorna la lista de candidaturas para la oferta con id *offerId*.

Caso de uso CU018

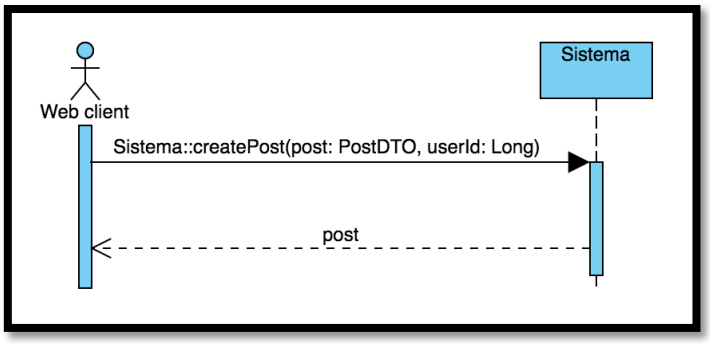


Ilustración 21: Diagrama de secuencia del caso de uso UC018

Contexto	Sistema::createPost(post: PostDTO, userId: Long)
Precondición	El usuario con id *userId* existe y la publicación proporcionada es válida.
Postcondición	Se retorna la publicación *post*, que ha sido creada en el sistema.

Caso de uso CU019

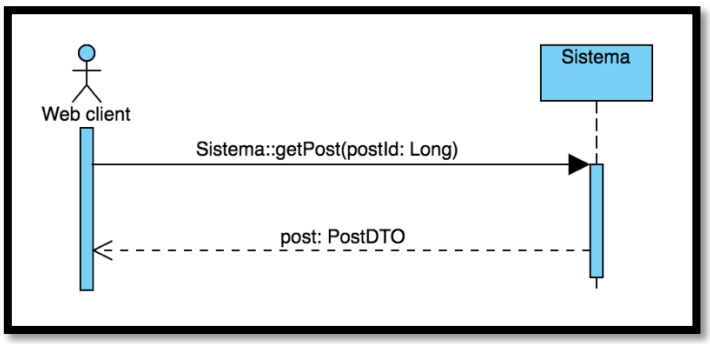


Ilustración 22: Diagrama de secuencia del caso de uso UC019

Contexto	Sistema::getPost(postId: Long)
Precondición	La publicación con id *postId* existe.
Postcondición	Se retorna la publicación con id *postId*.

Caso de uso CU020

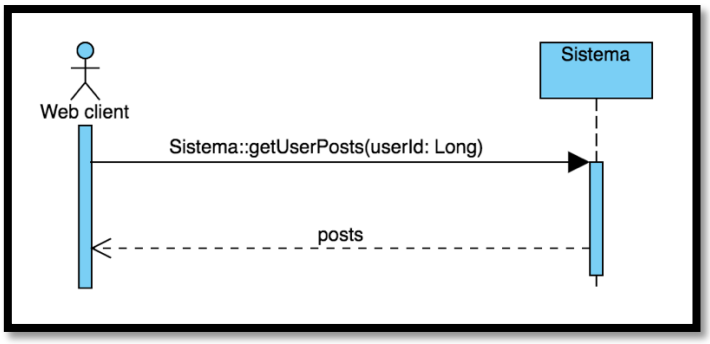


Ilustración 23: Diagrama de secuencia del caso de uso UC020

Contexto	Sistema::getUserPosts(userId: Long)
Precondición	El usuario con id *userId* existe.
Postcondición	Se retornan todas las publicaciones realizadas por el usuario especificado.

Caso de uso CU021

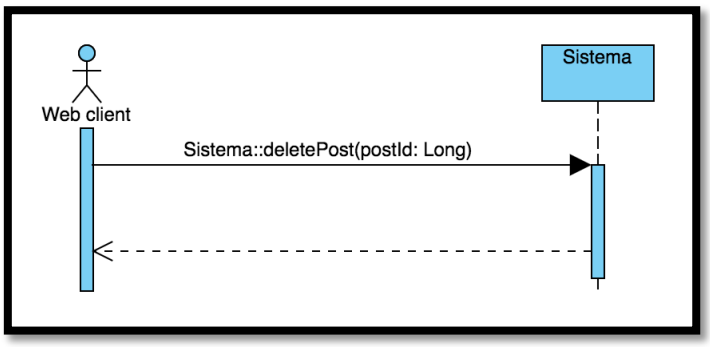


Ilustración 24: Diagrama de secuencia del caso de uso UC021

Contexto	Sistema::deletePost(postId: Long)
Precondición	La publicación con id *postId* existe.
Postcondición	La publicación con id *postId* ha sido eliminada del sistema.

Caso de uso CU022

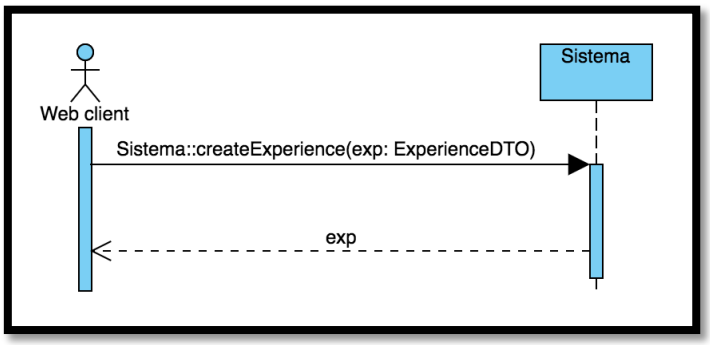


Ilustración 25: Diagrama de secuencia del caso de uso UC022

Contexto	Sistema::createExperience(exp: ExperienceDTO)
Precondición	La experiencia proporcionada es válida.
Postcondición	Se retorna la experiencia *exp*, que ha sido ha creada en el sistema.

Caso de uso CU023

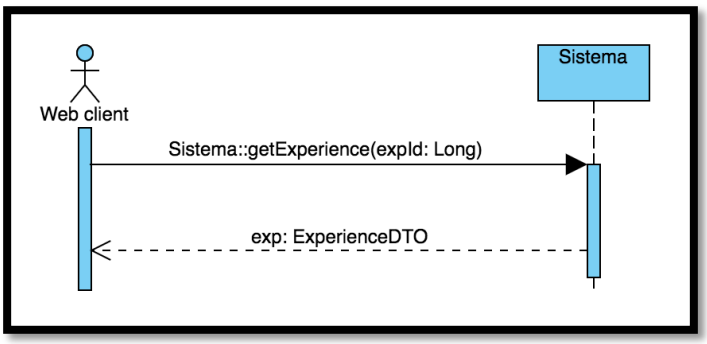


Ilustración 26: Diagrama de secuencia del caso de uso UC023

Contexto	Sistema::getExperience(expId: Long)
Precondición	La experiencia con id *expId* existe.
Postcondición	Se retorna la experiencia con id *expId*.

Caso de uso CU024

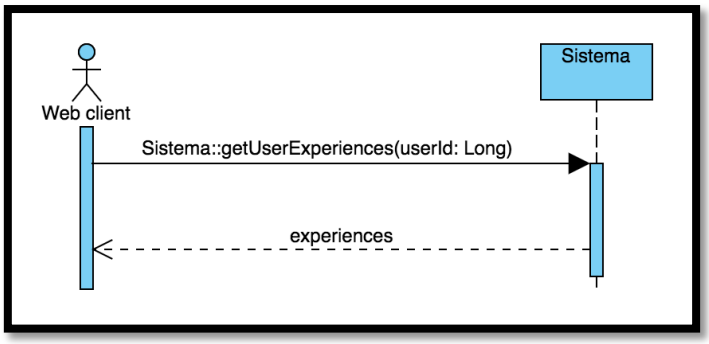


Ilustración 27: Diagrama de secuencia del caso de uso UC024

Contexto	Sistema::getUserExperiences(userId: Long)
Precondición	El usuario con id *userId* existe.
Postcondición	Se retornan todas las experiencias del usuario especificado.

Caso de uso CU025

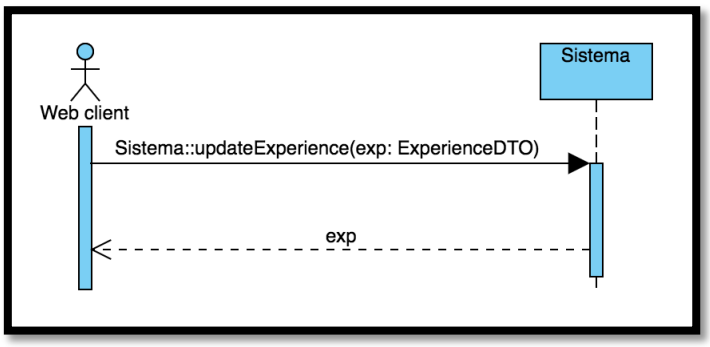


Ilustración 28: Diagrama de secuencia del caso de uso UC025

Contexto	Sistema::updateExperience(exp: ExperienceDTO)
Precondición	La experiencia proporcionada ya existe en el sistema y es válida.
Postcondición	Se retorna la experiencia *exp,* que ha sido modificada.

Caso de uso CU026

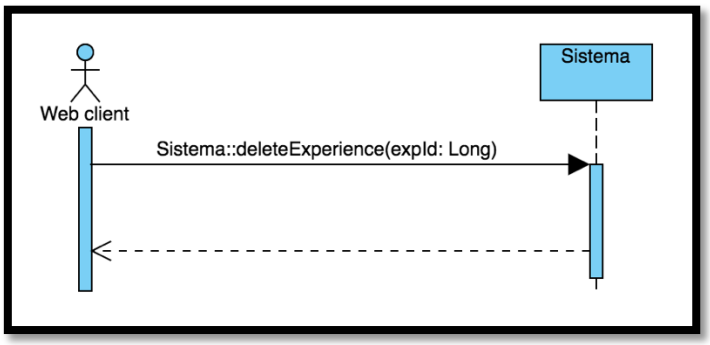


Ilustración 29: Diagrama de secuencia del caso de uso UC026

Contexto	Sistema::deleteExperience(expId: Long)
Precondición	La experiencia con id *expId* existe en el sistema.
Postcondición	La experiencia especificada ha sido eliminada.

7 Diseño del sistema

Una vez definidos los requisitos del sistema y realizado la especificación de cada uno de los casos de uso, es momento de proceder al diseño del sistema, que será el paso previo a su implementación.

Para poder llevarlo a cabo, se han tenido en cuenta buenas prácticas de programación además de utilizar arquitecturas y patrones de diseño ya conocidos que han facilitado la tarea.

A continuación se explican las propuestas de diseño realizadas, además de los aspectos que se han tenido en cuenta para llevarlas a cabo.

7.1 Arquitecturas de referencia

Para realizar el diseño de la arquitectura del sistema, se han tomado como referencia dos arquitecturas muy populares en la actualidad y que, guardando algunas diferencias, resultan muy parecidas.

A continuación se explican los principios básicos de cada una de ellas y los puntos en los que estas han influenciado el diseño del sistema.

7.1.1 Clean Architecture

La primera es la famosa *Clean architecture* de Robert Martin [14], que establece que la arquitectura de un sistema debería consistir en capas representadas como círculos concéntricos, donde aquellos más externos son los más concretos y los más cercanos al centro los más abstractos.

La capa exterior albergaría todo aquel código ligado a frameworks, específico de la plataforma para la que se desarrolla o el uso de servicios o bases de datos. La interior, por el contrario, contendría todas aquellas entidades necesarias para modelar el dominio del sistema. Por último, las

intermedias contendrían los casos de uso, que hacen uso de las entidades e implementan la lógica de negocio, y los controladores, que reciben las peticiones al sistema.

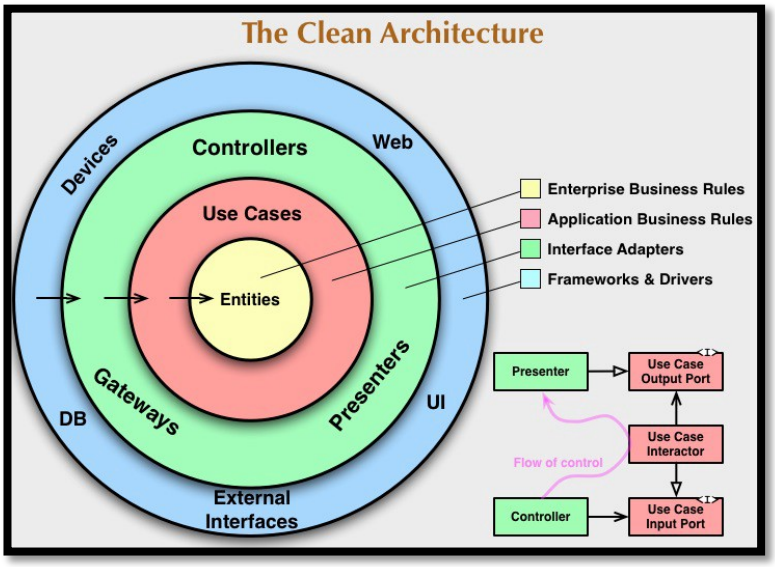


Ilustración 30: Diagrama de Clean Architecture

Otro concepto importante es la denominada regla de dependencias, que establece que una capa no puede tener conocimiento ni hacer uso de aquellas capas situadas más al exterior. De esta forma, se garantiza que al realizar un cambio, las capas interiores no se verán afectadas.

Mediante el uso de esta arquitectura, se genera un código fácil de probar y altamente cambiable, ya que debido al bajo acoplamiento entre capas, realizar pruebas y hacer cambios resulta extremadamente sencillo.

7.1.2 Arquitectura hexagonal

También se ha tomado como referencia la arquitectura hexagonal, también conocida como "*ports and adapters pattern*", ideada por Alistair Cockburn en el año 2005.

El principal objetivo de dicha arquitectura es la separación total del dominio de cualquier dependencia externa. Para ello, se utilizan dos conceptos clave:

- Puertos, que estandarizan las funcionalidades utilizadas de las dependencias. Gracias a ellos se consigue un total desacoplamiento que favorece la cambiabilidad y reusabilidad del sistema.

- Adaptadores, que son los encargados de adaptar las dependencias a las estructuras de datos definidas en el dominio del sistema.

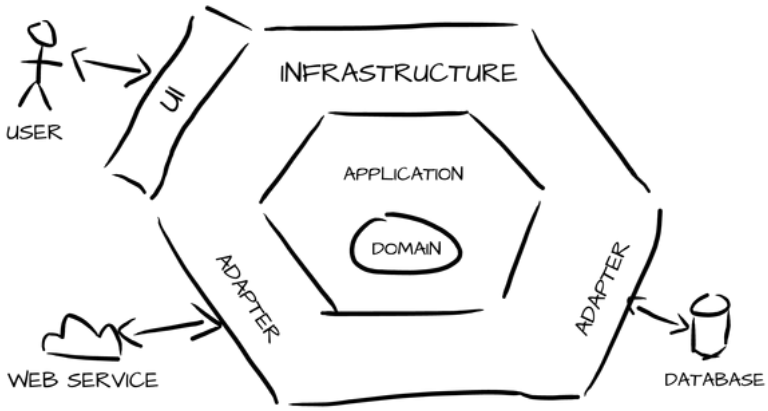


Ilustración 31: Diagrama de ejemplo de arquitectura hexagonal

Observando la Ilustración 31, se puede ver que esta guarda un parecido razonable con la anteriormente mencionada *Clean Architecture,* ya que en ambos se divide el código en diferentes capas desacoplando las entidades y la lógica de negocio de los detalles concretos de la implementación.

7.2 Patrones de diseño

Con el objetivo de obtener un mejor código, reducir el acoplamiento y solucionar algunos problemas comunes, se han utilizado una serie de patrones de diseño arquitectónicos, estructurales y de creación.

A continuación se resume el funcionamiento de cada uno de ellos así como la función que han desempeñado en el diseño del sistema **Error! Reference source not found.**.

7.2.1 Patrón *repository*

Este patrón añade una capa de abstracción entre la lógica de negocio de un sistema y la capa de acceso a datos. En ella se definen interfaces que, mediante una serie de métodos, permiten obtener o guardar objetos sin necesidad de conocer los detalles de las tecnologías o las fuentes de datos utilizadas.

En la siguiente ilustración, se puede observar como el repositorio se encarga de hacer las llamadas necesarias a dos fuentes de datos diferentes y de componer sus resultados, encapsulando toda la lógica de acceso a datos.

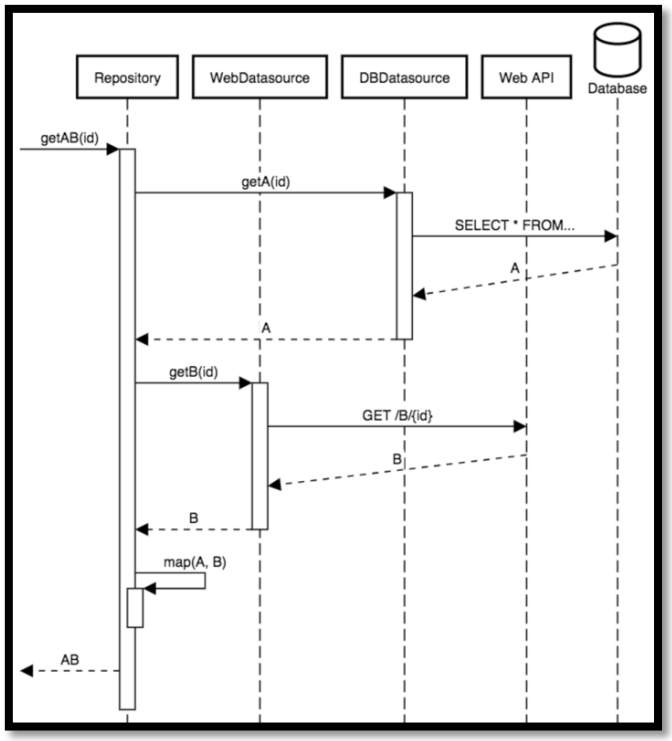


Ilustración 32: Ejemplo patrón repository

Para el diseño del sistema se ha creado un repositorio para cada tipo de objeto en el dominio, de modo que todos los accesos a datos se hicieran a través de ellos haciendo que, en caso de un cambio de tecnología, sólo sea necesario cambiar la implementación del repositorio.

7.2.2 Patrón *builder*

Este conocido patrón de diseño se encuentra dentro de la categoría de patrones creacionales y su función es la de controlar la creación de instancias de una clase. Este permite crear objetos complejos con diferentes partes que deben crearse en un orden determinado o utilizando un algoritmo específico.

Una clase externa, conocida como director, controla el algoritmo de construcción.

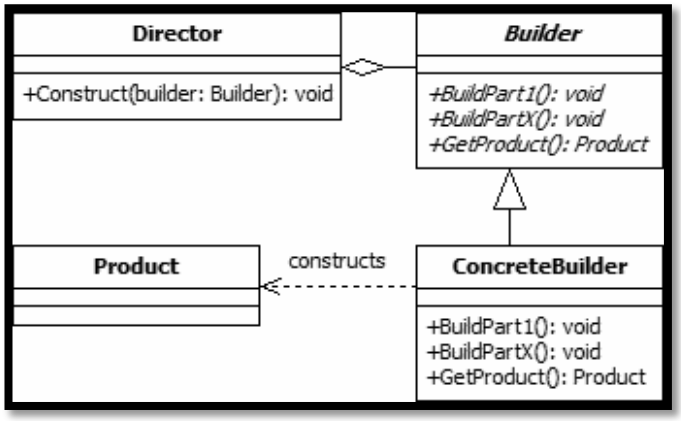


Ilustración 33: Ejemplo patrón builder

Se ha hecho uso de este patrón para la creación de todas las clases de dominio, definiendo un Builder en cada una de ellas y haciendo privado el constructor.

7.2.3 Patrón *prototype*

Este, al igual que el patrón *builder,* es considerado un patrón de creación ya que permite instanciar nuevos objetos a partir de otros ya existentes, copiando todas sus propiedades.

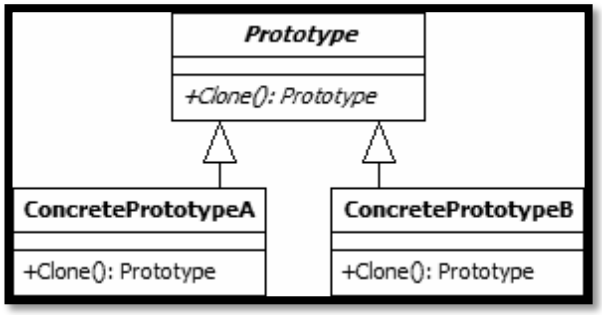


Ilustración 34: Ejemplo patrón prototype

De nuevo, se ha utilizado en todas las clases de dominio, añadiendo un método nuevo que, a partir de un objeto, genera una copia exacta.

7.2.4 Patrón *singleton*

El patrón *singleton* permite asegurar que únicamente se crea una instancia de un objeto determinado y que todas las referencias a dicho objeto se refieren a la misma instancia.

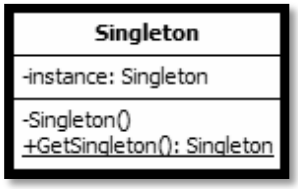


Ilustración 35: Ejemplo patrón singleton

Por defecto, *Spring* utiliza el patrón *singleton* para la creación de los *beans* definidos en los archivos de configuración, por lo que a menos que se especifique lo contrario, este asegurará que se crea una única instancia.

7.2.5 Patrón *API gateway*

En un sistema distribuido, realizar las peticiones directamente del cliente a los servicios conlleva ciertos problemas [21]:

- Las APIs podrían cambiar a medida que el sistema evoluciona, haciéndolas incompatibles con los clientes existentes.
- Podría cambiar la forma en la que los microservicios se estructuran, ya sea añadiendo nuevos o fusionando y eliminando los ya existentes.

- Algunos servicios podrían utilizar protocolos poco convencionales para un cliente, tales como *gRPC* o *AQMP*, utilizado por diferentes brókeres de eventos.

Estos pueden ser solucionados mediante el patrón *API Gateway,* que consiste en implementar un único punto de entrada al sistema, conocido como *edge-service*, que será el encargado de enrutar las peticiones al servicio correcto.

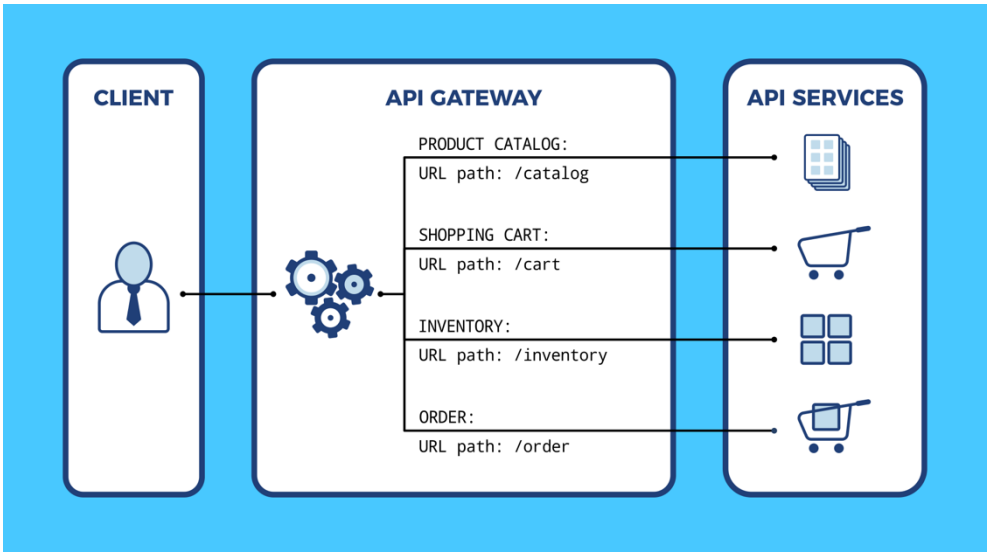


Ilustración 36: Ejemplo patrón API Gateway

Mediante la utilización de este patrón, se evita que los clientes tengan conocimiento de la estructura de los servicios del sistema, haciendo que esta pueda cambiar sin que se vean afectados.

7.2.6 Patrón *service registry*

En un sistema basado en microservicios, constantemente se crean o destruyen instancias dependiendo de la demanda. Esto hace que las direcciones IP y puertos cambien de forma dinámica, haciendo difícil conocer cuantas están disponibles y cómo acceder a ellas.

Para solucionar este problema se ha utilizado el patrón arquitectónico *service registry,* que permite llevar un registro de todas las instancias

disponibles en cada momento y de información acerca de ellas para más tarde ofrecérsela a aquellos servicios que la requieran.

Esto último se conoce como *service discovery* y hay dos formas diferentes de realizarlo [23]:

- ***Client-side service discovery:*** En este caso, es el cliente el encargado de realizar la petición para averiguar a qué instancia dirigirse. Para ello primero realizaría una llamada al *service registry* y luego directamente al servicio o al balanceador de carga, en caso de haberlo.

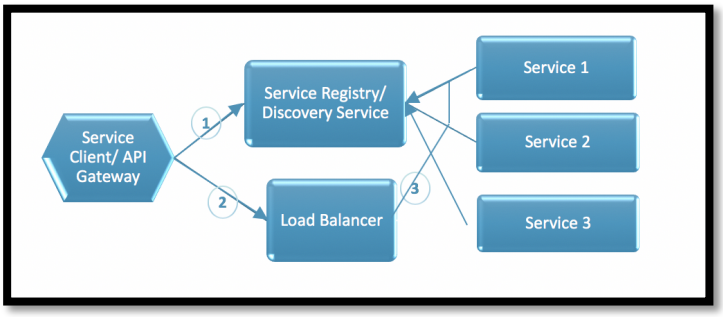


Ilustración 37: Service discovery en cliente

- ***Server-side service discovery***: En este caso, el encargado de realizar la petición al *service registry* no es el cliente, sino el *API Gateway*. El servicio realizará la petición directamente a este, que se encargará de obtener la información necesaria sobre la instancia para, posteriormente, enrutar la petición.

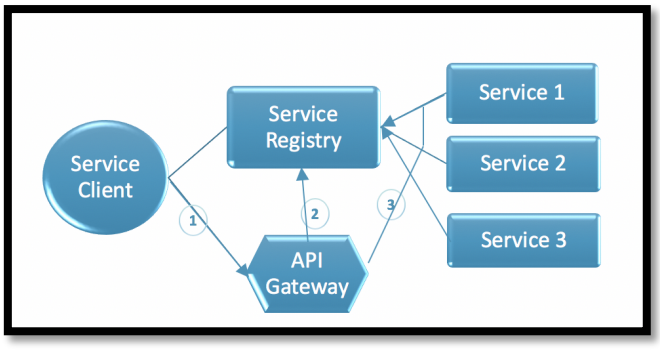


Ilustración 38: Service discovery en servidor

7.2.7 Patrón *circuit breaker*

Este patrón, muy popular en el desarrollo de sistemas distribuidos, consiste en encapsular la llamada a una función o un servicio dentro de un objeto *circuit breaker* que se encargará de monitorizar los fallos. Una vez los fallos superen cierto umbral el circuito se "abrirá" haciendo que todas las llamadas posteriores reciban directamente un error hasta haberse solucionado el problema [22].

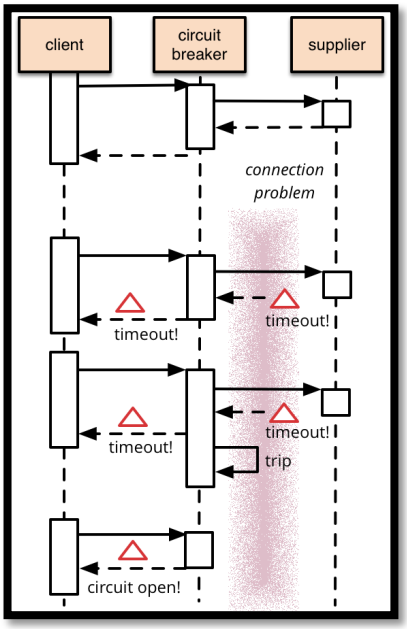


Ilustración 39: Ejemplo patrón circuit breaker

Gracias al uso de este patrón de diseño se puede evitar sobrecargar servicios con operaciones propensas a fallar. Por un lado evita al servicio que realiza la llamada tener que esperar hasta recibir un *timeout* y por el otro añadir más carga a un servidor saturado, ayudando así evitar fallos en cascada a través de los servicios.

7.3 Propuesta de diseño

Dado que este proyecto requiere de la implementación de dos sistemas distintos, se han realizado dos propuestas de diseño, una para cada uno de ellos, que servirán como guía a la hora de desarrollarlos.

7.3.1 Sistema monolítico

El diseño del sistema monolítico se basa en la arquitectura hexagonal, explicada en la sección 7.1.2. Los diferentes componentes del sistema se dividen en dos capas:

- **Dominio**: Contiene las entidades y toda la lógica de negocio, además de las interfaces que actuarán como puertos, de forma que el dominio se pueda comunicar con las dependencias.
- **Framework:** Contiene todo el código ligado al framework y a las dependencias del proyecto. Esta capa albergará principalmente los controladores, que recibirán peticiones HTTP, y los repositorios, que se encargarán de acceder a la base de datos.

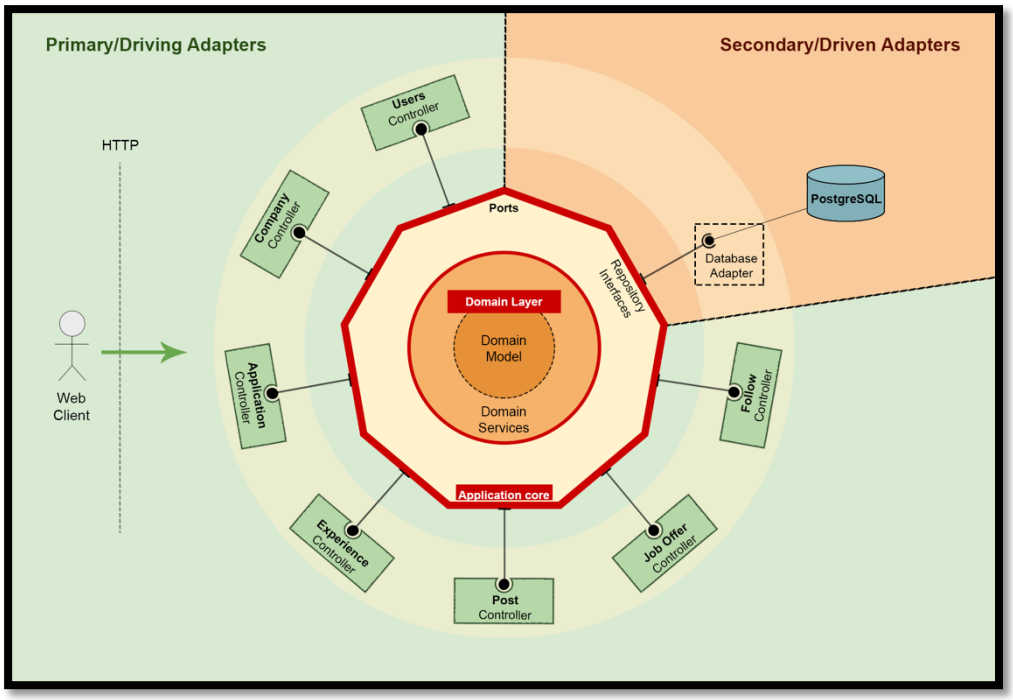


Ilustración 40: Diseño arquitectura monolítica

7.3.2 Sistema basado en microservicios

Al igual que en el caso del sistema monolítico, la propuesta de diseño del sistema basado en microservicios se basa en la arquitectura hexagonal, por lo que el diseño de cada microservicio será idéntico al del monolito.

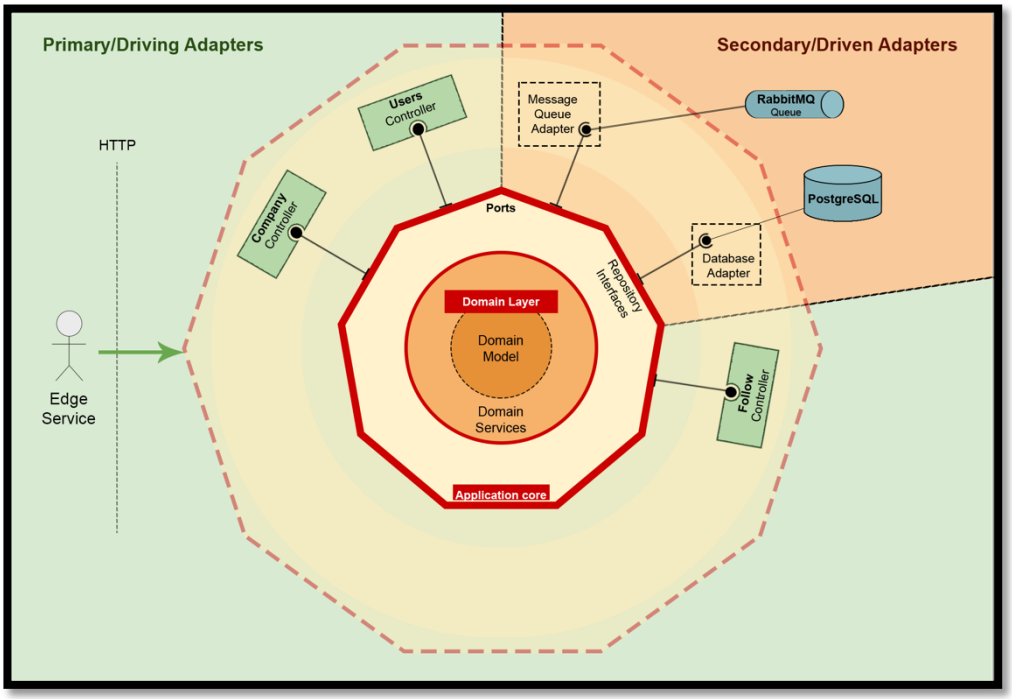


Ilustración 41: Diseño arquitectura microservicio

Sin embargo, dado que este es un sistema distribuido, será necesario definir también algunos detalles acerca de la infraestructura, que permitirá que estos trabajen de forma conjunta.

A continuación se listan los diferentes puntos que se han tenido en cuenta durante el diseño del sistema.

Comunicación interna

En determinadas ocasiones, será necesario que los diferentes servicios se comuniquen entre ellos, ya sea para realizar una petición o bien para notificar un cambio. Esto podría conseguirse mediante el uso de peticiones *HTTP*, aunque estas presentan varios problemas:

- Esta solución no es escalable, ya que cada servicio nuevo que se cree deberá añadirse para que este sea notificado también, lo cual puede complicarse en cuanto la complejidad del sistema aumente.
- Las peticiones se realizan de forma síncrona, lo que no resulta conveniente pues sería necesario esperar hasta que cada uno de los servicios respondiera.

Con el fin de solucionar todos estos problemas, se ha decidido hacer uso de un bróker de mensajes, como RabbitMQ, mediante el cual un servicio será capaz de publicar eventos en una cola de forma asíncrona, que mas tarde podrán ser consumidos por el resto de servicios del sistema.

Esta solución tampoco es perfecta, ya que el bróker no puede fallar o todo el sistema caería, aunque asegurando alta una disponibilidad esta resulta la mejor opción.

Comunicación externa

Otro punto importante a tener en cuenta es la forma en la que se accederá a los diferentes recursos desde fuera del sistema.

Realizar peticiones a los servicios directamente desde un cliente web o una aplicación móvil presenta ciertos inconvenientes como problemas de incompatibilidad al cambiar las APIs o reestructurar los microservicios. Sin embargo, estos pueden ser solucionados mediante el patrón API Gateway, explicado en el apartado 7.2.5.

Este patrón permitirá tener una única puerta de entrada y será el encargado de realizar el enrutamiento de las peticiones hacia los microservicios correspondientes, encapsulando de esta forma toda la lógica de acceso a los microservicios.

Para ello, se ha dispuesto un microservicio llamado *api-gateway* que deberá ser el único accesible desde internet y que, junto con el resto de microservicios, se encontrará dentro de una red privada mediante la cual podrá enviarles las peticiones que reciba.

Registro de servicios

Como se ha explicado en la sección 3.1, uno de los principales beneficios de las arquitecturas basadas en microservicios es la facilidad para escalar cada uno de los servicios de forma independiente.

Esto presenta un reto a la hora de realizar el diseño, pues será necesario llevar un control de cuantas instancias se encuentran disponibles de cada servicio y cuales son sus direcciones IP para que el resto de servicios puedan comunicarse con ellos.

Esto se conseguirá mediante la utilización del patrón *service registry,* explicado en el apartado 7.2.6. Para ello se creará un nuevo microservicio, el cual será notificado cada vez que se cree una nueva instancia y que permitirá al resto de servicios obtener dicha información cuando la requieran.

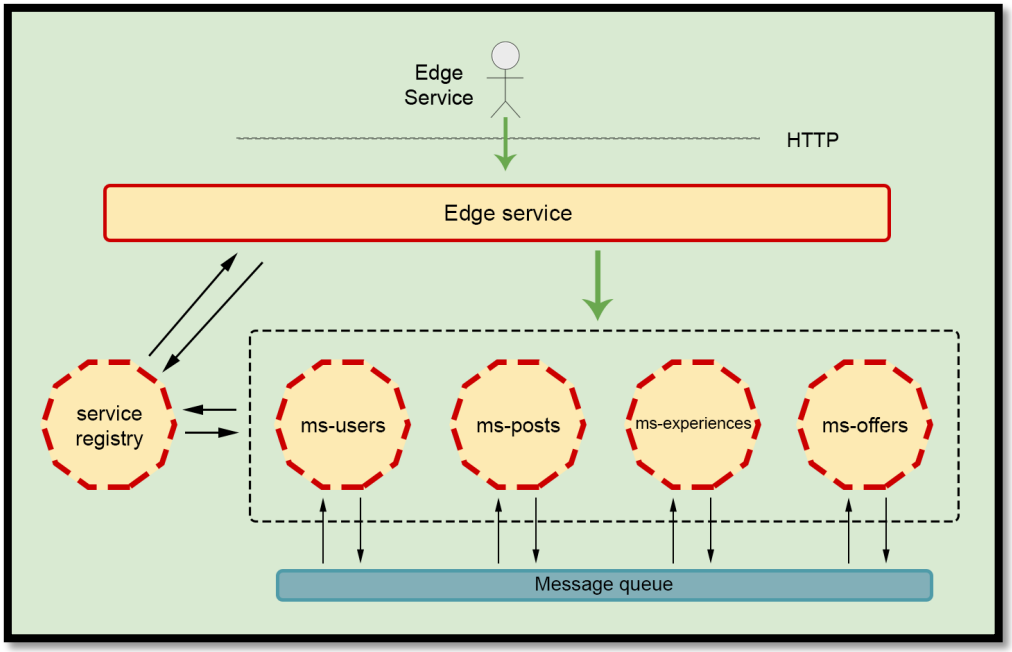


Ilustración 42: Diseño infraestructura microservicios

7.4 Especificación API

Al disponer de los casos de uso que deberá implementar el sistema, será necesario especificar los detalles de los diferentes *endpoints* que expondrán los diferentes servicios del sistema

Para ello, a continuación se detalla toda la información relevante acerca de estos, como el tiop de petición, la URL, los diferentes parámetros que deberán añadirse a la petición, los códigos de respuesta que puede devolver y un ejemplo de la respuesta, en caso de haberla.

Obtener todos los usuarios

GET	/users
Respuestas	
Código	Descripción
200	OK

Ejemplo respuesta	[
	  {
	  "birthdate": "2019-03-28T00:14:05.680Z",
	  "createdAt": "2019-03-28T00:14:05.680Z",
	  "id": 0,
	  "name": "string",
	  "surname": "string"
	  }
	]

Obtener usuario

GET	/users/{id}			
Parámetros				
Nombre	Tipo	Obligatorio	Descripción	Ejemplo
Id	Path	Si	Id de usuario	1000
Respuestas				
Código	**Descripción**			
200	OK			
404	Usuario no encontrado			

Ejemplo respuesta	{
	  "birthdate": "2019-03-28T00:14:05.710Z",
	  "createdAt": "2019-03-28T00:14:05.710Z",
	  "id": 1000,
	  "name": "string",
	  "surname": "string"
	}			

Crear usuario

POST	/users			
Parámetros				
Nombre	Tipo	Obligatorio	Descripción	Ejemplo

user	Body	Si	Usuario a crear	<pre>{
				  "birthdate": "2019-03-28T00:23:16.826Z",
				  "name": "string",
				  "surname": "string"
				}</pre>

Respuestas				
Código	**Descripción**			
200	OK			

Ejemplo respuesta	<pre>{
	  "birthdate": "2019-03-28T00:23:16.826Z",
	  "createdAt": "2019-03-28T00:23:16.826Z",
	  "id": 1000,
	  "name": "string",
	  "surname": "string"
	}</pre>			

Actualizar usuario

PUT	/users/{id}			
Parámetros				
Nombre	Tipo	Obligatorio	Descripción	Ejemplo

user	Body	Si	Usuario a actualizar	<pre>{
				  "birthdate": "2019-03-28T00:23:16.826Z",
				  "name": "string",
				  "surname": "string"
				}</pre>

id	Path	Si	Id de usuario	1000
Respuestas				
Código	**Descripción**			
200	OK			
404	Usuario no encontrado			

Ejemplo respuesta	<pre>{
	  "birthdate": "2019-03-28T00:23:16.826Z",
	  "createdAt": "2019-03-28T00:23:16.826Z",
	  "id": 1000,
	  "name": "string",
	  "surname": "string"
	}</pre>			

Borrar usuario

DELETE	/users/{id}			
Parámetros				
Nombre	Tipo	Obligatorio	Descripción	Ejemplo
Id	Path	Si	Id de usuario	1000
Respuestas				
Código	**Descripción**			
200	OK			
404	Usuario no encontrado			
Ejemplo respuesta	-			

Obtener todas las empresas

GET	/companies
Respuestas	
Código	Descripción
200	OK

Ejemplo respuesta	<pre>[
	  {
	  "createdAt": "2019-03-28T17:10:15.306Z",
	  "description": "string",
	  "id": 1000,
	  "name": "string"
	  }
	]</pre>

Obtener empresa

GET	/companies/{id}			
Parámetros				
Nombre	Tipo	Obligatorio	Descripción	Ejemplo
Id	Path	Si	Id de empresa	1000
Respuestas				
Código	**Descripción**			
200	OK			
404	Empresa no encontrada			

Ejemplo respuesta	<pre>{
	  "createdAt": "2019-03-28T17:10:15.306Z",
	  "description": "string",
	  "id": 1000,
	  "name": "string"
	}</pre>			

Crear empresa

POST	/companies			
Parámetros				
Nombre	Tipo	Obligatorio	Descripción	Ejemplo

company	Body	Si	Empresa a crear	<pre>{
				  "description": "string",
				  "name": "string"
				}</pre>

Respuestas				
Código	**Descripción**			
200	OK			

Ejemplo respuesta	<pre>{
	  "createdAt": "2019-03-28T17:10:15.306Z",
	  "description": "string",
	  "id": 1000,
	  "name": "string"
	}</pre>			

Actualizar empresa

PUT	/companies/{id}			
Parámetros				
Nombre	Tipo	Obligatorio	Descripción	Ejemplo

company	Body	Si	Empresa a actualizar	<pre>{
				  "description": "string",
				  "name": "string"
				}</pre>

id	Path	Si	Id de empresa	1000
Respuestas				
Código	**Descripción**			
200	OK			
404	Empresa no encontrada			

Ejemplo respuesta	<pre>{
	  "createdAt": "2019-03-28T17:10:15.306Z",
	  "description": "string",
	  "id": 1000,
	  "name": "string"
	}</pre>			

Eliminar empresa

DELETE	/companies/{id}			
Parámetros				
Nombre	Tipo	Obligatorio	Descripción	Ejemplo
Id	Path	Si	Id de empresa	1000
Respuestas				
Código	**Descripción**			
200	OK			
404	Empresa no encontrada			
Ejemplo respuesta	-			

Obtener experiencias académicas de un usuario

GET	/education-experiences			
Parámetros				
Nombre	Tipo	Obligatorio	Descripción	Ejemplo
userId	Param	Si	Id de usuario	1000
Respuestas				
Código	Descripción			
200	OK			

Ejemplo respuesta	<pre>[
	  {
	  "dateFrom": "2019-03-28T17:43:34.328Z",
	  "dateTo": "2019-03-28T17:43:34.328Z",
	  "degree": "string",
	  "description": "string",
	  "field": "string",
	  "grade": "string",
	  "id": 0,
	  "school": "string",
	  "userId": 0
	  }
	]</pre>			

Obtener experiencia académica

GET	/education-experiences/{id}			
Parámetros				
Nombre	Tipo	Obligatorio	Descripción	Ejemplo
Id	Path	Si	Id de experiencia	1000
Respuestas				
Código	**Descripción**			
200	OK			
404	Experiencia no encontrada			

Ejemplo respuesta	{
	"dateFrom": "2019-03-28T17:43:34.328Z",
	"dateTo": "2019-03-28T17:43:34.328Z",
	"degree": "string",
	"description": "string",
	"field": "string",
	"grade": "string",
	"id": 0,
	"school": "string",
	"userId": 0
	}			

Crear experiencia académica

POST	/education-experiences			
Parámetros				
Nombre	Tipo	Obligatorio	Descripción	Ejemplo

experience	Body	Si	Experiencia a crear	{
				"dateFrom": "2019-03-28T17:43:34.328Z",
				"dateTo": "2019-03-28T17:43:34.328Z",
				"degree": "string",
				"description": "string",
				"field": "string",
				"grade": "string",
				"school": "string",
				"userId": 0
				}

Respuestas				
Código	**Descripción**			
200	OK			
500	Usuario no encontrado			

Ejemplo respuesta	{
	"id": "1000",
	"dateFrom": "2019-03-28T17:43:34.328Z",
	"dateTo": "2019-03-28T17:43:34.328Z",
	"degree": "string",
	"description": "string",
	"field": "string",
	"grade": "string",
	"school": "string",
	"userId": 0
	}			

Actualizar experiencia académica

PUT	/education-experiences/{id}			
Parámetros				
Nombre	Tipo	Obligatorio	Descripción	Ejemplo

experience	Body	Si	Experiencia a actualizar	<pre>{
				  "dateFrom": "2019-03-28T17:43:34.328Z",
				  "dateTo": "2019-03-28T17:43:34.328Z",
				  "degree": "string",
				  "description": "string",
				  "field": "string",
				  "grade": "string",
				  "school": "string",
				  "userId": 0
				}</pre>

id	Path	Si	Id experiencia	1000
Respuestas				
Código	**Descripción**			
200	OK			
404	Usuario o experiencia no encontrados			

Ejemplo respuesta	<pre>{
	  "id": "1000",
	  "dateFrom": "2019-03-28T17:43:34.328Z",
	  "dateTo": "2019-03-28T17:43:34.328Z",
	  "degree": "string",
	  "description": "string",
	  "field": "string",
	  "grade": "string",
	  "school": "string",
	  "userId": 0
	}</pre>			

Eliminar experiencia académica

DELETE	/education-experiences/{id}			
Parámetros				
Nombre	Tipo	Obligatorio	Descripción	Ejemplo
Id	Path	Si	Id de experiencia	1000
Respuestas				
Código	**Descripción**			
200	OK			
404	Experiencia no encontrada			
Ejemplo respuesta	-			

Obtener experiencias laborales de un usuario

GET	/work-experiences			
Parámetros				
Nombre	Tipo	Obligatorio	Descripción	Ejemplo
userId	Param	Si	Id de usuario	1000
Respuestas				
Código	Descripción			
200	OK			
Ejemplo respuesta	`[ { "company": "string", "dateFrom": "2019-03-28T17:53:25.417Z", "dateTo": "2019-03-28T17:53:25.417Z", "description": "string", "id": 0, "industry": "string", "location": "string", "title": "string", "userId": 0 } ]`			

Obtener experiencia laboral

GET	/work-experiences/{id}			
Parámetros				
Nombre	Tipo	Obligatorio	Descripción	Ejemplo
Id	Path	Si	Id de experiencia	1000
Respuestas				
Código	**Descripción**			
200	OK			
404	Experiencia no encontrada			
Ejemplo respuesta	`{ "company": "string", "dateFrom": "2019-03-28T17:53:25.417Z", "dateTo": "2019-03-28T17:53:25.417Z", "description": "string", "id": 0, "industry": "string", "location": "string", "title": "string", "userId": 0 }`			

Crear experiencia laboral

POST	/work-experiences			
Parámetros				
Nombre	Tipo	Obligatorio	Descripción	Ejemplo

experience	Body	Si	Experiencia a crear	<pre>{
				"company": "string",
				"dateFrom": "2019-03-28T17:53:25.417Z",
				"dateTo": "2019-03-28T17:53:25.417Z",
				"description": "string",
				"industry": "string",
				"location": "string",
				"title": "string",
				"userId": 0
				}</pre>

Respuestas				
Código	**Descripción**			
200	OK			
500	Usuario no encontrado			

Ejemplo respuesta	<pre>{
	"id": "1000",
	"dateFrom": "2019-03-28T17:43:34.328Z",
	"dateTo": "2019-03-28T17:43:34.328Z",
	"degree": "string",
	"description": "string",
	"field": "string",
	"grade": "string",
	"school": "string",
	"userId": 0
	}</pre>			

Actualizar experiencia laboral

PUT	/work-experiences/{id}			
Parámetros				
Nombre	Tipo	Obligatorio	Descripción	Ejemplo

experience	Body	Si	Experiencia a actualizar	<pre>{
				"company": "string",
				"dateFrom": "2019-03-28T17:53:25.417Z",
				"dateTo": "2019-03-28T17:53:25.417Z",
				"description": "string",
				"industry": "string",
				"location": "string",
				"title": "string",
				"userId": 0
				}</pre>

id	Path	Si	Id de experiencia	1000
Respuestas				
Código	**Descripción**			
200	OK			
404	Usuario o experiencia no encontrados			

Ejemplo respuesta	<pre>{
	"company": "string",
	"dateFrom": "2019-03-28T17:53:25.417Z",
	"dateTo": "2019-03-28T17:53:25.417Z",
	"description": "string",
	"id": 0,
	"industry": "string",
	"location": "string",
	"title": "string",
	"userId": 0}</pre>			

Eliminar experiencia laboral

DELETE	/work-experiences/{id}			
Parámetros				
Nombre	Tipo	Obligatorio	Descripción	Ejemplo
Id	Path	Si	Id de experiencia	1000
Respuestas				
Código	**Descripción**			
200	OK			
404	Experiencia no encontrada			
Ejemplo respuesta	-			

Obtener publicaciones de un usuario

GET	/posts			
Parámetros				
Nombre	Tipo	Obligatorio	Descripción	Ejemplo
userId	Path	Si	Id de usuario	1000
Respuestas				
Código	Descripción			
200	OK			

Ejemplo respuesta	[
	{
	"id": 0,
	"publicationDate": "2019-03-28T18:00:09.433Z",
	"text": "string",
	"userId": 1000
	}
	]			

Obtener publicación

GET	/posts/{id}			
Parámetros				
Nombre	Tipo	Obligatorio	Descripción	Ejemplo
Id	Path	Si	Id de publicación	1000
Respuestas				
Código	**Descripción**			
200	OK			
404	Publicación no encontrada			

Ejemplo respuesta	{
	"id": 0,
	"publicationDate": "2019-03-28T18:00:55.537Z",
	"text": "string",
	"userId": 1000
	}			

Crear publicación

POST	/posts			
Parámetros				
Nombre	Tipo	Obligatorio	Descripción	Ejemplo

Publicación	Body	Si	Publicación a crear	<pre>{
				  "text": "string",
				  "userId": 0
				}</pre>

Respuestas				
Código	**Descripción**			
200	OK			
500	Usuario no existe			

Ejemplo respuesta	<pre>{
	  "id": 0,
	  "publicationDate": "2019-03-28T18:00:55.537Z",
	  "text": "string",
	  "userId": 0
	}</pre>			

Eliminar publicación

DELETE	/posts/{id}			
Parámetros				
Nombre	Tipo	Obligatorio	Descripción	Ejemplo
Id	Path	Si	Id de publicación	1000
Respuestas				
Código	**Descripción**			
200	OK			
404	Publicación no encontrada			
Ejemplo respuesta	-			

Obtener ofertas de trabajo de una empresa

GET	/job-offers			
Parámetros				
Nombre	Tipo	Obligatorio	Descripción	Ejemplo
companyId	Param	Si	Id de empresa	1000
Respuestas				
Código	Descripción			
200	OK			
Ejemplo respuesta	`[{ "companyId": 0, "createdAt": "2019-03-28T18:38:22.270Z", "description": "string", "employmentType": "string", "id": 0, "industry": "string", "jobFunctions": ["string"], "location": "string", "requiredSkills": ["string"], "salary": 0, "title": "string", "updatedAt": "2019-03-28T18:38:22.270Z" }]`			

Obtener oferta de trabajo

GET	/job-offers/{id}			
Parámetros				
Nombre	Tipo	Obligatorio	Descripción	Ejemplo
Id	Path	Si	Id de oferta	1000
Respuestas				
Código	**Descripción**			
200	OK			
404	Oferta no encontrada			
Ejemplo respuesta	`{ "companyId": 0, "createdAt": "2019-03-28T18:51:44.501Z", "description": "string", "employmentType": "string", "id": 0, "industry": "string", "jobFunctions": [ "string" ], "location": "string", "requiredSkills": [ "string" ], "salary": 0, "title": "string", "updatedAt": "2019-03-28T18:51:44.501Z" }`			

Crear oferta de trabajo

POST	/job-offers

Parámetros

Nombre	Tipo	Obligatorio	Descripción	Ejemplo

offer	Body	Si	Oferta a crear	<pre>{
				  "companyId": 0,
				  "description": "string",
				  "employmentType": "string",
				  "industry": "string",
				  "jobFunctions": [
				  "string"
				  ],
				  "location": "string",
				  "requiredSkills": [
				  "string"
				  ],
				  "salary": 0,
				  "title": "string",
				}</pre>

Respuestas

Código	Descripción
200	OK
500	Empresa no encontrado

Ejemplo respuesta

<pre>{
  "companyId": 0,
  "createdAt": "2019-03-28T18:51:44.501Z",
  "description": "string",
  "employmentType": "string",
  "id": 0,
  "industry": "string",
  "jobFunctions": [
  "string"
  ],
  "location": "string",
  "requiredSkills": [
  "string"
  ],
  "salary": 0,
  "title": "string",
  "updatedAt": "2019-03-28T18:51:44.501Z"
}</pre>

Actualizar oferta de trabajo

PUT	/job-offers/{id}			
Parámetros				
Nombre	Tipo	Obligatorio	Descripción	Ejemplo

offer	Body	Si	Oferta a actualizar	<pre>{
				  "companyId": 0,
				  "description": "string",
				  "employmentType": "string",
				  "industry": "string",
				  "jobFunctions": [
				  "string"
				  ],
				  "location": "string",
				  "requiredSkills": [
				  "string"
				  ],
				  "salary": 0,
				  "title": "string",
				}</pre>

id	Path	Si	Id de oferta	1000
Respuestas				
Código	**Descripción**			
200	OK			
404	Empresa u oferta no encontrados			

Ejemplo respuesta	<pre>{
	  "companyId": 0,
	  "createdAt": "2019-03-28T18:51:44.501Z",
	  "description": "string",
	  "employmentType": "string",
	  "id": 0,
	  "industry": "string",
	  "jobFunctions": [
	  "string"
	  ],
	  "location": "string",
	  "requiredSkills": [
	  "string"
	  ],
	  "salary": 0,
	  "title": "string",
	  "updatedAt": "2019-03-28T18:51:44.501Z"
	}</pre>			

Eliminar oferta de trabajo

DELETE	/job-offers/{id}			
Parámetros				
Nombre	Tipo	Obligatorio	Descripción	Ejemplo
Id	Path	Si	Id de oferta	1000
Respuestas				
Código	**Descripción**			
200	OK			
404	Oferta no encontrada			
Ejemplo respuesta	-			

Obtener candidaturas de una oferta

GET	/applications			
Parámetros				
Nombre	Tipo	Obligatorio	Descripción	Ejemplo
offerId	Param	Si	Id de oferta	1000
Respuestas				
Código	Descripción			
200	OK			
Ejemplo respuesta	`[ { "applicationDate": "2019-03-28T19:00:45.998Z", "coverLetter": "string", "id": 0, "offerId": 0, "userId": 0 } ]`			

Obtener candidatura

GET	/applications/{id}			
Parámetros				
Nombre	Tipo	Obligatorio	Descripción	Ejemplo
Id	Path	Si	Id de oferta	1000
Respuestas				
Código	**Descripción**			
200	OK			
404	Oferta no encontrada			
Ejemplo respuesta	`{ "applicationDate": "2019-03-28T19:04:58.806Z", "coverLetter": "string", "id": 0, "offerId": 0, "userId": 0 }`			

Crear candidatura

POST	/applications			
Parámetros				
Nombre	Tipo	Obligatorio	Descripción	Ejemplo

application	Body	Si	Candidatura a crear	<pre>{
				  "coverLetter": "string",
				  "offerId": 0,
				  "userId": 0
				}</pre>

Respuestas				
Código	**Descripción**			
200	OK			
500	Usuario u oferta no encontrados			

Ejemplo respuesta	<pre>{
	  "applicationDate": "2019-03-28T19:04:58.806Z",
	  "coverLetter": "string",
	  "id": 0,
	  "offerId": 0,
	  "userId": 0
	}</pre>			

Actualizar candidatura

PUT	/applications/{id}			
Parámetros				
Nombre	Tipo	Obligatorio	Descripción	Ejemplo

offer	Body	Si	Oferta a actualizar	<pre>{
				  "coverLetter": "string",
				  "offerId": 0,
				  "userId": 0
				}</pre>

id	Path	Si	Id de oferta	1000
Respuestas				
Código	**Descripción**			
200	OK			
404	Usuario, oferta o candidatura no encontrados			

Ejemplo respuesta	<pre>{
	  "applicationDate": "2019-03-28T19:04:58.806Z",
	  "coverLetter": "string",
	  "id": 0,
	  "offerId": 0,
	  "userId": 0
	}</pre>			

Eliminar candidatura

DELETE	/applications/{id}			
Parámetros				
Nombre	Tipo	Obligatorio	Descripción	Ejemplo
Id	Path	Si	Id de candidatura	1000
Respuestas				
Código	**Descripción**			
200	OK			
404	Candidatura no encontrada			
Ejemplo respuesta	-			

Obtener seguidores de un usuario

GET	/users/{id}/followers			
Parámetros				
Nombre	Tipo	Obligatorio	Descripción	Ejemplo
offerId	Param	Si	Id de usuario	1000
Respuestas				
Código	Descripción			
200	OK			
404	Usuario no encontrado			

Ejemplo respuesta	[
	{
	"createdAt": "2019-03-28T19:14:51.693Z",
	"user": {
	"birthdate": "2019-03-28T19:14:51.693Z",
	"createdAt": "2019-03-28T19:14:51.693Z",
	"id": 0,
	"name": "string",
	"surname": "string"
	}
	}
	]			

Obtener usuarios seguidos por usuario

GET	/users/{id}/following			
Parámetros				
Nombre	Tipo	Obligatorio	Descripción	Ejemplo
Id	Path	Si	Id de usuario	1000
Respuestas				
Código	**Descripción**			
200	OK			
404	Usuario no encontrado			
Ejemplo respuesta	`[ { "createdAt": "2019-03-28T19:14:51.693Z", "user": { "birthdate": "2019-03-28T19:14:51.693Z", "createdAt": "2019-03-28T19:14:51.693Z", "id": 0, "name": "string", "surname": "string" } } ]`			

Seguir a usuario

POST	/users/{id}/following			
Parámetros				
Nombre	Tipo	Obligatorio	Descripción	Ejemplo
id	Body	Si	Id de usuario seguidor	1000
userId	Param	Si	Id usuario seguido	1000
Respuestas				
Código	**Descripción**			
200	OK			
500	Usuario u oferta no encontrados			
Ejemplo respuesta	`{ "createdAt": "2019-03-28T19:11:59.625Z", "user": { "birthdate": "2019-03-28T19:11:59.625Z", "createdAt": "2019-03-28T19:11:59.625Z", "id": 0, "name": "string", "surname": "string" } }`			

Dejar de seguir a usuario

DELETE		/users/{id}/following		
Parámetros				
Nombre	Tipo	Obligatorio	Descripción	Ejemplo
Id	Path	Si	Id de usuario seguidor	1000
userId	Param	Si	Id de usuario seguido	1000
Respuestas				
Código		**Descripción**		
200		OK		
Ejemplo respuesta		-		

8 Entorno de desarrollo

8.1 Lenguajes de programación

8.1.1 Java

Para el desarrollo del proyecto se ha utilizado Java como lenguaje principal. Este es un lenguaje orientado a objetos basado en clases, de uso general, concurrente y fuertemente tipado [19].

La principal ventaja respecto al resto de lenguajes y el motivo por el que se ha escogido este lenguaje para llevar a cabo la implementación de ambos sistemas es el gran número de bibliotecas disponibles que permiten aplicar patrones comunes para sistemas distribuidos, explicados en el apartado 3.3.

Ilustración 43: Ejemplo lenguaje Java

8.1.2 Groovy

Apache Groovy es un lenguaje potente, opcionalmente tipado y dinámico, con capacidades de compilación y tipado estáticas. Se integra sin problemas con cualquier programa Java e inmediatamente ofrece a la aplicación funciones que incluyen capacidades de scripting, creación de lenguajes específicos del dominio, programación en tiempo de ejecución y metacompilación y programación funcional.

Este lenguaje se utiliza en los archivos de configuración de *gradle*, donde se especifican las dependencias del proyecto así como diferentes tareas de *gradle*, entre otras funcionalidades.

```
task printProperties {
    doLast {
        println springVersion
        println emailNotification
        sourceSets.matching { it.purpose == "production" }.each { println it.name }
    }
}
```

Ilustración 44: Ejemplo lenguaje Groovy

8.1.3 Yaml

YAML o '*Yet Another Markup Language*' es un lenguaje de serialización cuya popularidad ha aumentado constantemente en los últimos años. A menudo se usa como formato para archivos de configuración, pero sus capacidades de serialización de objetos lo convierten en un reemplazo viable para lenguajes como JSON [20].

En el caso de este proyecto, *YAML* ha sido utilizado únicamente para los archivos de configuración de los proyectos *Spring*.

```
spring:
  datasource:
    url: "jdbc:postgresql://${POSTGRES_ADDRESS}:5432/postgres"
    username: gvalentin
    password: ${POSTGRES_PASSWORD}
  jpa:
    properties:
      hibernate:
        temp:
          use_jdbc_metadata_defaults: false
```

Ilustración 45: Ejemplo lenguaje YAML

8.2 Bibliotecas y plugins externos

8.2.1 Lombok

Lombok es una biblioteca desarrollada en Java que permite generar código de forma automática en tiempo de compilación haciendo uso únicamente de anotaciones.

Esto no solo agiliza el desarrollo si no que también favorece la legibilidad del código, ya que evita implementar ciertas funcionalidades comunes, reduciendo el número de líneas de código.

Entre las diferentes opciones que ofrece, se encuentran algunas como autogenerado de *getters* y *setters*, constructores o *builders*.

```java
@Getter
@AllArgsConstructor
@Builder
public class User {
    private final String name;
    private final String surname;
}
```

Ilustración 46: Ejemplo biblioteca Lombok

8.2.2 Swagger UI

Swagger UI permite generar automáticamente una web con documentación interactiva sobre los recursos ofrecidos por las diferentes APIs de un sistema, además de permitir visualizar e interactuar con ellas sin necesidad de implementar la lógica.

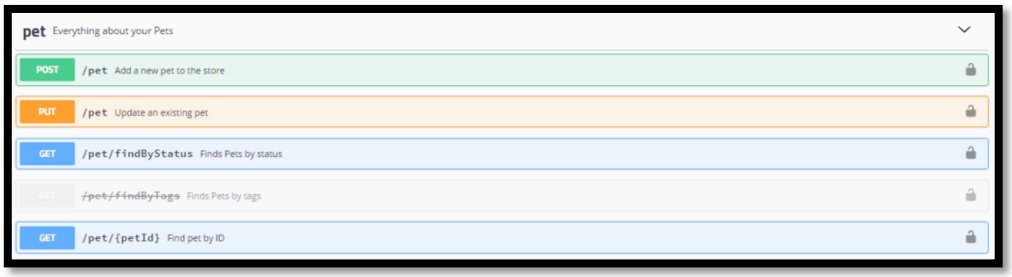


Ilustración 47: Ejemplo Swagger UI

Este ha sido utilizado para ver de forma rápida un listado de los *endpoints* del sistema así como para realizar algunas pruebas.

8.2.3 Flyway

Flyway es una biblioteca que permite realizar migraciones incrementales en bases de datos. Esta se ha utilizado para ejecutar los *scripts* mediante los cuales se realiza la creación de tablas y relaciones en la base de datos utilizada para los tests de integración.

8.2.4 DBUnit

DBUnit es una extensión de la conocida biblioteca *JUnit*, pero enfocada a base de datos. Se ha utilizado para comprobar el estado inicial y final de la base de datos en los tests de integración y verificar que estos funcionaban correctamente.

Su funcionamiento es muy sencillo, ya que para llevar a cabo estas verificaciones únicamente es necesario añadir algunas anotaciones en los tests, además de crear ficheros *XML* con el contenido esperado de la base de datos.

8.2.5 RestAssured

RestAssured es una de las bibliotecas más populares para realizar tests automatizados de *APIs.* Esta se ha utilizado en los test de integración para simular peticiones HTTP al sistema y verificar después la respuesta obtenida.

```
@Test
public void whenValidateResponseTime_thenSuccess() {
    when().get("/users/eugenp").then().time(lessThan(5000L));
}
```

Ilustración 48: Ejemplo de uso de RestAssured

8.3 Framework

Para el desarrollo, tanto del sistema monolítico como del basado en microservicios, se ha utilizado *Spring Boot,* uno de los frameworks más populares en la actualidad.

Este presenta una gran versatilidad, ya que ofrece una serie de funcionalidades básicas que se pueden ampliar mediante el uso de módulos específicos en función de las necesidades.

Además, permite integrar de forma sencilla las herramientas desarrolladas por *Netflix* para gestionar una infraestructura de microservicios, lo cual lo hace una perfecta alternativa para el desarrollo de este tipo de sistemas.

A continuación, se detallan algunos de los módulos utilizados para el desarrollo de este proyecto.

8.3.1 Spring MVC

Spring MVC es un framework web que permite a la aplicación recibir peticiones HTTP. Para lograr esto, define un *Servlet* central, llamado *DispatcherServlet* que proporciona un algoritmo compartido para gestionar peticiones a través de los controladores.

Su funcionamiento es muy sencillo gracias a la autoconfiguración de *Spring Boot,* por lo que para hacerlo funcionar únicamente será necesario añadir la dependencia en el gestor de dependencias, *Gradle* en el caso de este proyecto, y comenzar a definir controladores.

```
implementation 'org.springframework.boot:spring-boot-starter-web'
```

Ilustración 49: Dependencia Spring MVC en gradle

Para poder recibir peticiones desde un controlador será necesario añadir la anotación *@RestController* a la clase así como *@RequestMapping,* de forma opcional, que permitirá definir la *URL* base.

Una vez hecho esto se podrán utilizar anotaciones como *@GetMapping, @PostMapping* y *@DeleteMapping*, entre otras, que asegurarán que peticiones GET, POST y DELETE, respectivamente, son mapeadas a los métodos que las incluyan.

Spring MVC también permite realizar un mapeo de parámetros, ya sea dentro del *body* de la petición o en la url, así como de objetos. Para ello se utilizarán las anotaciones *@RequestParam, @PathVariable y @RequestBody.*

En la siguiente ilustración se puede observar un pequeño controlador de ejemplo donde se utilizan todas las anotaciones mencionadas anteriormente.

```
@RestController
@RequestMapping("/users")
public class UserController {

    @PostMapping
    @ResponseBody
    public UserDto createUser(@RequestBody UserDto user) { ... }

    @GetMapping("/{id}")
    @ResponseBody
    public UserDto getUser(@PathVariable("id") Long id) { ... }

    @DeleteMapping
    public void deleteUser(@RequestParam(name="id") Long id) { ... }

}
```

Ilustración 50: Ejemplo de controlador

8.3.2 Spring Data JPA

Spring Data JPA es uno de los componentes de Spring Data cuyo objetivo principal es el de abstraer al desarrollador de la implementación de la capa de datos.

```
implementation 'org.springframework.boot:spring-boot-starter-data-jpa'
```

Ilustración 51: Dependencia Spring Data JPA en gradle

De nuevo, para empezar a hacer uso de este módulo, bastará con añadir la dependencia a gradle.

Una vez añadida será necesario añadir la configuración del *datasource* en el archivo de properties correspondiente.

```yaml
spring:
  datasource:
    driver-class-name: org.postgresql.Driver
    url: "jdbc:postgresql://localhost:12100/postgres"
    username: "postgres"
    password: "123456"
  jpa:
    properties:
      hibernate:
        dialect: org.hibernate.dialect.PostgreSQLDialect
    database-platform: org.hibernate.dialect.PostgreSQLDialect
```

Ilustración 52: Ejemplo de configuración de un datasource

En la ilustración anterior se puede observar un ejemplo de configuración donde se define el datasource mediante la especificación del *driver* y de la dirección y credenciales de la base de datos.

Aunque no es necesario, también se especifica el dialecto a utilizar. En este caso *PostgreSQL,* ya que esta es la base de datos que se usará.

Spring Data JPA también permite la implementación automática de repositorios mediante el uso de la interfaz *CrudRepository.* Con ello, se conseguirá de forma instantánea un repositorio con una serie de queries por defecto tales como *save, delete, count o findAll,* entre otras. Además, también ofrece otras funcionalidades tales como predicados *QueryDsl* o generación automática de *queries* a partir del nombre del método definido en el repositorio.

En la siguiente ilustración se puede observar un ejemplo de la definición de un repositorio, que *Spring* implementará automáticamente a partir de la sintaxis del método definido.

```java
public interface UserRepository extends CrudRepository<UserEntity, Long> {
    public List<User> findAllByIdWhereNameEquals(String name);
}
```

Ilustración 53: Repositorio de ejemplo

Por último, *Spring Data JPA* también incorpora *Hibernate*, cuyas anotaciones permiten definir tablas, columnas, relaciones y *constraints* de

manera sencilla. En la siguiente ilustración se muestra un ejemplo de entidad implementada en el proyecto.

```java
@Entity
@Table(
        name = "applications",
        uniqueConstraints = @UniqueConstraint(columnNames={"user_id", "offer_id"})
)
public class ApplicationEntity {

    @Id
    @GeneratedValue(generator = "applications_generator")
    @SequenceGenerator(
            name = "applications_generator",
            sequenceName = "applications_seq",
            initialValue = 1000,
            allocationSize = 1
    )
    private Long id;

    @ManyToOne
    @OnDelete(action = OnDeleteAction.CASCADE)
    @JoinColumn(name="user_id", updatable = false)
    private UserEntity user;

    @ManyToOne
    @OnDelete(action = OnDeleteAction.CASCADE)
    @JoinColumn(name="offer_id", updatable = false)
    private JobOfferEntity offer;
```

Ilustración 54: Entidad de ejemplo

8.3.3 Spring Cloud

Spring Cloud es un proyecto de *Spring* que engloba multitud de herramientas cuyo objetivo principal es el de facilitar la creación y mantenimiento de sistemas distribuidos.

Para el desarrollo de este trabajo, se han utilizado tres de las herramientas de *Spring Cloud*:

Spring Cloud Netflix

Spring Cloud Netflix provee integración con *Netflix OSS*, un conjunto de herramientas open-source creadas por *Netflix* y extensamente reconocidas como una de las mejores alternativas para configurar la infraestructura de microservicios.

Estas permiten activar, mediante anotaciones, patrones comunes en sistemas distribuidos tales como *Service Discovery*, *Circuit breaker* o *Distributed sessions*, entre muchos otros. En la sección 3.3 se listan las herramientas más conocidas.

Spring Cloud Stream

Spring Cloud Stream es un framework que permite crear microservicios *event-driven* mediante la implementación de un sistema de eventos de forma sencilla gracias al uso de anotaciones.

Este facilita la integración con sistemas como *Apache Kafka* o *RabbitMQ,* entre otros, de forma que los microservicios puedan producir y consumir eventos para comunicarse entre ellos.

Los tres conceptos básicos necesarios para entender el funcionamiento de *Spring Cloud Stream* son los siguientes:

- **Destination Binders**: Componentes responsables de la integración con el sistema de mensajería externo.
- **Destination Bindings**: Puente entre los sistemas de mensajería externos y la aplicación, proporcionado por los *producers* y *consumers* y creado por el *destination binder*.
- **Mensajes**: La estructura de datos canónica usada por los *producers* y *consumers* para comunicarse con el *destination binder*.

El primer paso para poder utilizar *Spring Cloud Stream* sería añadir su dependencia a gradle junto con la del binder a utilizar, por ejemplo el de RabbitMQ. Como alternativa, también se puede utilizar la dependencia de la Ilustración 55, que ya incluye todo lo necesario.

```
implementation 'org.springframework.cloud:spring-cloud-starter-stream-rabbit'
```

Ilustración 55: Dependencia Spring cloud stream starter

Una vez hecho esto, ya se podrán crear *bindings* mediante la anotación *@EnableBinding*, la cual notifica al *framework* para que cree la cola de

mensajes, el *topic*, etc. Para ello también será necesario especificar un canal y si este es de entrada o de salida. A continuación se puede observar un ejemplo de creación de un *binding*:

```
@Configuration
@EnableBinding({UserChannel.class})
public class MessagingConfiguration {
```

Ilustración 56: Ejemplo de creación de un binding

```
public interface UserChannel {

    String INPUT = "tfg.users";

    @Input(UserChannel.INPUT)
    SubscribableChannel input();

}
```

Ilustración 57: Ejemplo de creación de un canal de entrada

Para finalizar, Spring Cloud Stream ofrece la anotación *@StreamListener*, que mapeará todos los eventos recibidos al método que la incorpore. Para ello será necesario especificar el canal al que subscribirse. En la siguiente ilustración se puede observar un consumer de ejemplo.

```
public class UserMessageConsumer {

    @StreamListener(UserChannel.INPUT)
    private void handle(CustomMessage message) {
        // Este método se ejecutará cada ve que se
        // reciba un nuevo evento.
    }
}
```

Ilustración 58: Ejemplo de creación de un consumer

8.4 Herramientas

8.4.1 Gradle

Gradle es un sistema de automatización de compilación de código abierto que se basa en los conceptos de Apache Ant y Apache Maven e introduce un lenguaje específico de dominio (DSL) basado en Groovy en lugar del formulario XML utilizado por Apache Maven para declarar la configuración del proyecto. Este utiliza un grafo acíclico dirigido para determinar el orden en que se pueden ejecutar las tareas.

Este admite construcciones incrementales al determinar de manera inteligente qué partes del árbol de compilación están actualizadas.

Se ha utilizado tanto en el monolito como en el sistema basado en microservicios para gestionar las dependencias así como para la definición de tareas.

8.4.2 Docker

Docker es un proyecto open source que utiliza la virtualización a nivel de sistema operativo para desarrollar y entregar software en paquetes llamados contenedores, una opción mucho más ligera que las máquinas virtuales ya que estos se ejecutan en un solo kernel de sistema operativo.

Los contenedores están aislados entre sí y agrupan su propio software, bibliotecas y archivos de configuración, aunque pueden comunicarse entre sí a través de canales bien definidos. Esto facilita el despliegue de aplicaciones, ya que todas las dependencias y configuración se realiza dentro del contenedor, que podrá ejecutarse en cualquier máquina siempre y cuando esta cumpla los requisitos mínimos.

Tanto el monolito como los diferentes servicios del sistema basado en microservicios, se han introducido dentro de un contenedor *alpine,* una versión ligera de Linux, para poder ser desplegados tanto en local como en *AWS* sin ninguna configuración previa.

8.4.3 RabbitMQ

RabbitMQ es un software de cola de mensajes o *message broker* basado en el protocolo *AMQP* que permite la creación de colas de mensajes donde aplicaciones pueden conectarse y producir mensajes, así como consumirlos.

Este se ha utilizado en el sistema basado en microservicios para permitir la comunicación asíncrona entre servicios de forma sencilla.

9 Desarrollo del sistema

Una vez acabados el diseño, la especificación de requisitos y de haber elegido las tecnologías que se usarán en el proyecto, es el momento de realizar el desarrollo de ambos sistemas.

En esta sección se explicarán los detalles de la implementación del monolito y del sistema basado en microservicios, desde la estructura del proyecto hasta el uso de los patrones previamente explicados.

9.1 Sistema monolítico

9.1.1 Estructura del proyecto

El proyecto se ha dividido en dos capas, *dominio* y *framework*. El primero consiste en entidades (*model*) y lógica de negocio (*service*), así como interfaces que actúan como puertos para que la capa de framework pueda interactuar con el dominio. El segundo contiene todo el código vinculado al framework y dependencias, que se encargará de recibir peticiones HTTP (*controller*), o de hacer consultas sobre la base de datos (*repository*).

En la siguiente ilustración se puede observar el diagrama de paquetes del proyecto, que muestra la estructura del mismo.

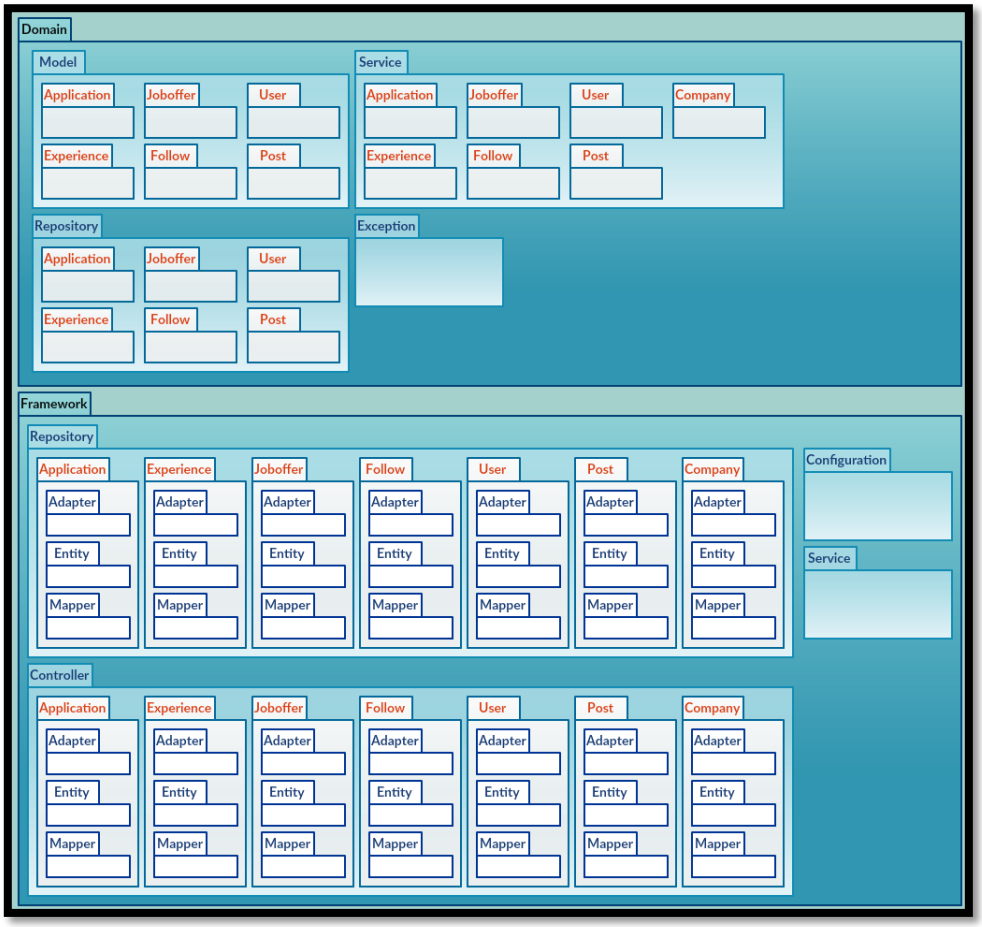


Ilustración 59: Diagrama de paquetes del sistema monolítico

Como se puede observar, se ha decidido realizar una fragmentación horizontal del código (por tipos de clases, como modelos, servicios o repositorios), aunque una vez finalizado el desarrollo se llegó a la conclusión de que hubiera sido mejor hacerla vertical (por funcionalidades).

Esto es debido a que, actualmente, para añadir, modificar o eliminar una funcionalidad, habría que cambiar muchos paquetes. Sin embargo, el particionado vertical hubiera favorecido este tipo de cambios, ya que únicamente hubiera hecho falta añadir o eliminar un paquete con todas las clases de esa funcionalidad.

9.1.2 Implementación de un caso de uso

Con el fin de mostrar el flujo completo de la ejecución de un caso de uso y dado que todos son muy similares, se explicará con detalle uno de ellos.

La siguiente ilustración muestra un diagrama de secuencia con el flujo de ejecución del caso de uso UC001, que consiste en crear un usuario. En él se puede observar como llega una petición al controlador y como, mediante un adaptador, este se comunica con el servicio que está en dominio.

A continuación se muestran fragmentos de código para conocer con más detalle en que consiste cada componente.

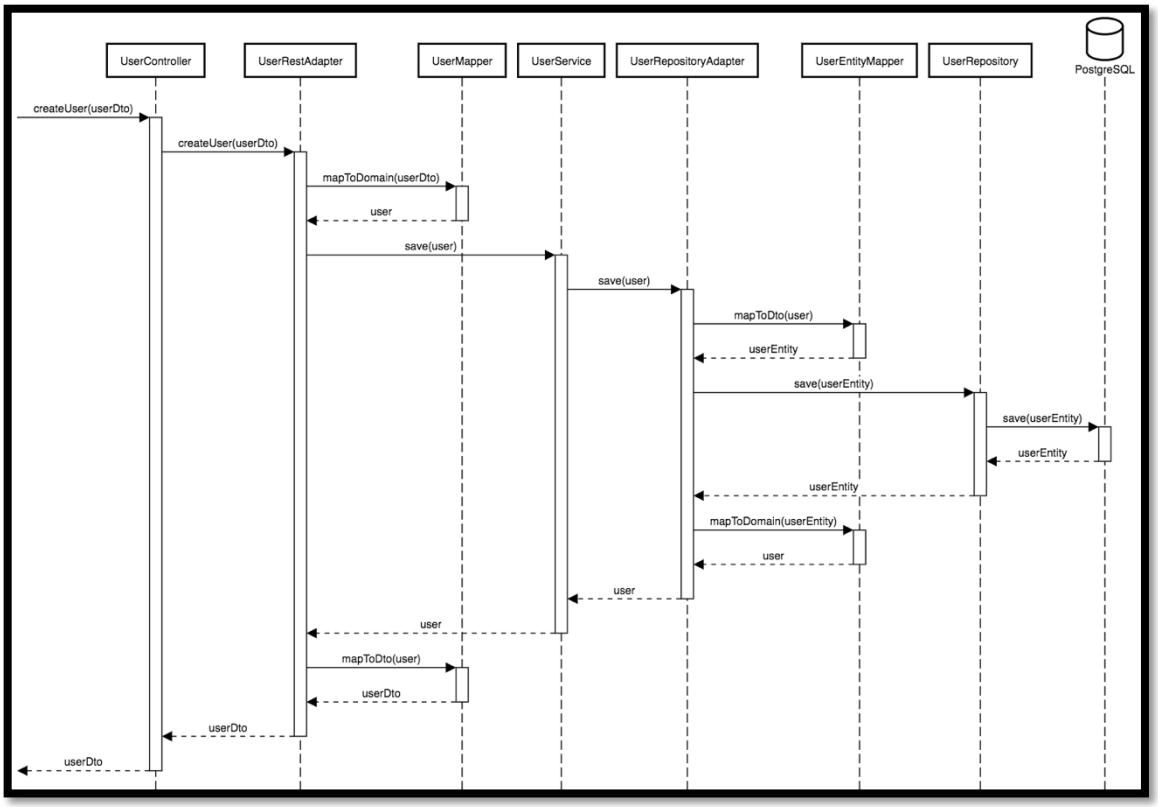


Ilustración 60: Diagrama de secuencia de UC001

Controladores

Para implementar los controladores, se ha utilizado la anotación *@RestController*, explicada en la sección 8.3.1, mediante la cual Spring mapeará todas las peticiones HTTP realizadas a la ruta especificada en la anotación *@RequestMapping.*

A nivel de método, se han utilizado las anotaciones *@PostMapping, @PutMapping, @GetMapping* y *@DeleteMapping* que permitirán mapear las peticiones en función de su tipo a la función correspondiente.

```
@RestController
@RequestMapping("/users")
public class UserController {

    @Autowired
    private UserRestAdapter restAdapter;

    @PostMapping
    @ResponseBody
    public UserDto createUser(@RequestBody UserDto user) { return restAdapter.createUser(user); }

    @PutMapping
    @ResponseBody
    public UserDto updateUser(@RequestBody UserDto user) { return restAdapter.updateUser(user); }

    @GetMapping
    @ResponseBody
    public List<UserDto> getUsers() { return restAdapter.getUsers(); }

    @GetMapping("/{id}")
    @ResponseBody
    public UserDto getUser(@PathVariable("id") Long id) { return restAdapter.getUser(id); }

    @DeleteMapping("/{id}")
    public void deleteUser(@PathVariable("id") Long id) { restAdapter.deleteUser(id); }

}
```

Ilustración 61: Controlador usuarios

La función de los controladores es la de recibir las peticiones y realizar las llamadas al caso de uso correspondiente mediante el adaptador *rest.*

Adaptadores Rest

Se ha denominado adaptador *rest* a una capa intermedia añadida entre los controladores y la lógica de negocio cuyo objetivo principal es el de adaptar los objetos utilizados en el controlador a objetos de dominio antes de realizar la llamada al servicio. Para ello, estos hacen uso del *mapper* correspondiente en función del caso de uso.

```java
public class UserRestAdapter {

    @Autowired
    private UserMapper userMapper;

    @Autowired
    private UserService userService;

    public UserDto createUser(UserDto dto) {
        return userMapper.mapToDto(userService.createUser(userMapper.mapToDomain(dto)));
    }

    public UserDto updateUser(UserDto dto) {
        return userMapper.mapToDto(userService.updateUser(userMapper.mapToDomain(dto)));
    }

    public List<UserDto> getUsers() {
        return userService.getUsers().stream().map(userMapper::mapToDto).collect(Collectors.toList());
    }

    public UserDto getUser(Long id) { return userMapper.mapToDto(userService.getUser(id)); }

    public void deleteUser(Long id) { userService.deleteUser(id); }
}
```

Ilustración 62: Adaptador rest usuarios

En la ilustración anterior se puede observar un adaptador *rest* que, en el caso de crear un usuario, se encarga de realizar la llamada y adaptar tanto el objeto que le pasa al servicio como el objeto resultante.

Mappers

Los *mappers* son clases cuyo único objetivo es el de convertir objetos de un tipo a otro. Dado que todos deberían tener las mismas funcionalidades, se creó una interfaz que definiera los dos únicos métodos que deberían implementar, utilizando los tipos genéricos de Java.

```java
public interface Mapper<T, S> {
    S mapToDto(T model);
    T mapToDomain(S dto);
}
```

Ilustración 63: Interfaz mappers

En la siguiente ilustración se puede observar un *mapper*, en este caso utilizado para convertir objetos *User* a *UserDTO* y viceversa, cuya

implementación consiste únicamente en copiar campo por campo el contenido del objeto.

```java
public class UserMapper implements Mapper<User, UserDto> {

    @Override
    public UserDto mapToDto(User model) {
        return UserDto.newBuilder()
                .withId(model.getId())
                .withName(model.getName())
                .withSurname(model.getSurname())
                .withBirthdate(model.getBirthdate())
                .withCreatedAt(model.getCreatedAt())
                .build();
    }

    @Override
    public User mapToDomain(UserDto dto) {
        return User.newBuilder()
                .withId(dto.getId())
                .withName(dto.getName())
                .withSurname(dto.getSurname())
                .withBirthdate(dto.getBirthdate())
                .withCreatedAt(dto.getCreatedAt())
                .build();
    }

}
```

Ilustración 64: Mapper de usuarios

Se han creado un *mappers* por cada *adapter*, tanto *rest* como de respositorios.

Servicios

Los servicios contienen la lógica de negocio. En el caso de este proyecto, esto consiste únicamente en realizar las llamadas a la base de datos para guardar o consultar objetos pues no es necesario hacer ningún otro tipo de cálculo o transformación de los datos.

La siguiente ilustración muestra el servicio de usuarios, que contiene los diferentes casos de uso vinculados a estos.

```java
@Service
public class UserService {

    @Autowired
    private UserRepositoryAdapter repository;

    public User createUser(User user) { return repository.save(user); }

    public User updateUser(User user) { return repository.update(user); }

    public User getUser(Long id) { return repository.findById(id); }

    public void deleteUser(Long id) { repository.deleteById(id); }

    public List<User> getUsers() { return repository.findAll(); }
}
```

Ilustración 65: Servicio de usuarios

Repositorios

Los repositorios permiten abstraer al dominio de las tecnologías utilizadas para persistir los datos. Estos se definen mediante una interfaz en el dominio y, en caso de ser necesario, se implementan en la parte de *framework*.

```java
public interface FollowRepositoryAdapter {
    void unfollow(Long followerId, Long followedId);
    List<Follow> getFollowers(Long id);
    List<Follow> getFollowed(Long id);
    Follow follow(Long followerId, Long followedId);
}
```

Ilustración 66: Repositorio de follows

En el caso de este proyecto, se han utilizado únicamente las funcionalidades que ofrecen los repositorios generados por *Spring JPA*, por lo que no ha sido necesario realizar la implementación de ninguno de ellos.

Como se explica en la sección 8.3.2, Spring se encarga de autogenerar la implementación de los repositorios a partir de la configuración ubicada en el archivo *application.yml* del proyecto. A demás de las funcionalidades por defecto, en la Ilustración 66 se puede observar como se han definido algunos métodos extra que también serán implementados de forma automática a partir de la sintaxis de los nombres.

Entidades

En el contexto de este proyecto, se denomina entidad al objeto que la base de datos conoce. Cada objeto de dominio tiene una entidad, que definirá como se persistirá en la base de datos.

Como se explicó en la sección 8.3.2, Spring Data facilita la tarea con un conjunto de anotaciones que permiten definir que campos actúan como clave primaria, que relaciones tiene una determinada clase con el resto o incluso definir diferentes restricciones.

Mediante estas anotaciones, Spring será capaz de autogenerar el esquema de la base de datos, con las tablas, columnas y relaciones especificadas.

```java
@Table(name = "job_offers")
public class JobOfferEntity {

    @Id
    @GeneratedValue(generator = "job_offers_generator")
    @SequenceGenerator(
            name = "job_offers_generator",
            sequenceName = "job_offers_seq",
            initialValue = 1000,
            allocationSize = 1
    )
    private Long id;
    private String title;
    private String description;
    private String employmentType;
    private String industry;
    private String location;
    private Integer salary;

    @CreationTimestamp
    @Column(updatable = false)
    private Instant createdAt;

    @UpdateTimestamp
    private Instant updatedAt;

    @ElementCollection
    @OnDelete(action = OnDeleteAction.CASCADE)
    @CollectionTable(name = "job_offer_job_functions", joinColumns = @JoinColumn(name = "job_offer_id"))
    @JoinColumn(name = "job_offer_id")
    private List<String> jobFunctions;

    @ElementCollection
    @OnDelete(action = OnDeleteAction.CASCADE)
    @CollectionTable(name = "job_offer_required_skills", joinColumns = @JoinColumn(name = "job_offer_id"))
    @JoinColumn(name = "job_offer_id")
    private List<String> requiredSkills;

    @ManyToOne
    @OnDelete(action = OnDeleteAction.CASCADE)
    private CompanyEntity company;
```

Ilustración 67: Ejemplo de entidad

En la ilustración anterior se puede observar una entidad existente en el sistema donde mediante la anotación *@Id* se ha definido que ese campo será clave primaria y donde haciendo uso únicamente de las anotaciones

@ElemenCollection y @CollectionTable, automáticamente se generará una tabla donde se guardarán el conjunto de funciones o habilidades vinculadas a la oferta.

9.1.3 Esquema base de datos

Una vez definidas todas las entidades del sistema, el esquema de la base de datos se ha generado junto con todas las tablas y relaciones.

En la siguiente ilustración se puede ver el diagrama de datos.

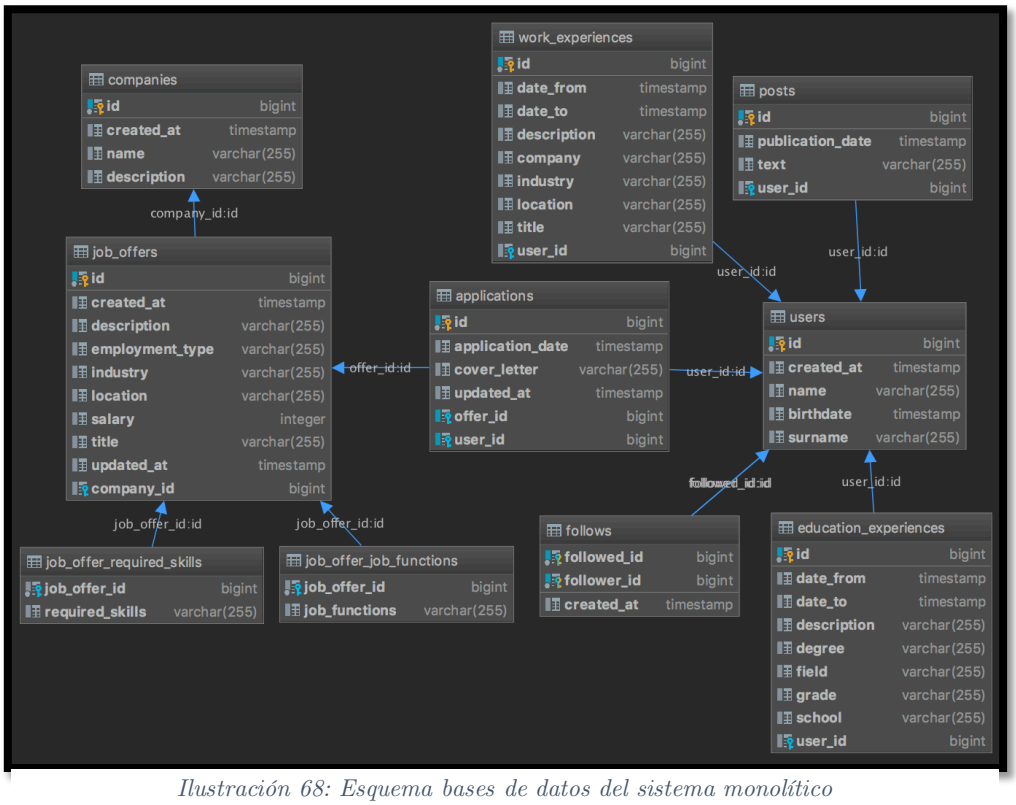


Ilustración 68: Esquema bases de datos del sistema monolítico

9.2 Sistema basado en microservicios

El desarrollo de los microservicios resulta muy parecido al del sistema monolítico, aunque al ser un sistema distribuido no solo es necesario implementar los diferentes casos de uso de cada servicio, si no que también

hay que configurar toda una infraestructura para permitir que estos puedan recibir peticiones y comunicarse entre ellos.

A continuación se explica cual ha sido el procedimiento para crear cada uno de los componentes de este sistema.

9.2.1 Estructura de los servicios

La estructura de cada servicio es idéntica a la utilizada en el sistema monolítico. Esta se compone de dos paquetes principales: dominio y framework.

El primero contiene todos los modelos de dominio así como la lógica de negocio y los diferentes casos de uso. El segundo contiene todo el código ligado al framework y a las diferentes tecnologías utilizadas, como bases de datos, colas de mensajes o controladores REST.

9.2.2 Servicios de infraestructura

Service registry

Como se explica en el apartado 7.2.6, el *service registry* es el encargado de llevar un control de las instancias disponibles de cada uno de los servicios. Para ello se ha utilizado *Eureka*, una de las herramientas creadas por *Netflix* y disponibles dentro del framework *Spring Cloud.*

El primer paso para integrar esta herramienta dentro del sistema ha sido crear un nuevo servicio al que se ha añadido la anotación *@EnableEurekaServer* en la clase principal junto con la anotación @SpringBootApplication típica de las aplicaciones realizadas con este framework.

```
@EnableEurekaServer
@SpringBootApplication
public class EurekaServerApplication {
    public static void main(String[] args) {
        SpringApplication.run(EurekaServerApplication.class, args);
    }
}
```

Ilustración 69: Anotaciones requeridas por el service registry

Una vez hecho esto, el servicio está listo para funcionar como un servidor *Eureka*, aunque todavía es necesario añadir algún parámetro de configuración al *application.yml*.

La siguiente ilustración muestra un ejemplo de configuración donde se especifica el puerto por el que enviará y recibirá peticiones, el nombre de la instancia y que no es necesario que se registre con el *service registry*, ya que es él mismo.

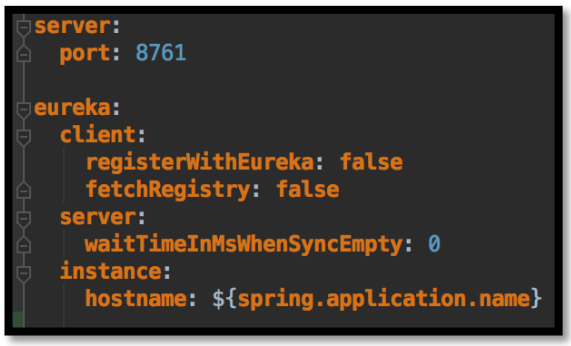


Ilustración 70: Configuración del service registry

Con estos dos pasos el nuevo servicio está listo para comenzar a registrar servicios, que deberán dirigirse a la dirección *http://{ip-servicio}:8761/eureka*.

Edge service

El *edge service* es un servicio que actúa como puerta de entrada y salida de la infraestructura de microservicios, tal y como dicta el patrón *API Gateway* explicado en el apartado 7.2.5.

De nuevo, esto se ha conseguido mediante la creación de un nuevo servicio al que se han añadido las anotaciones *@EnableZuulProxy* y *@EnableEurekaClient*, que permiten que este utilice Zuul para enrutar las peticiones recibidas así como que se conecte a Eureka.

Esto último permitirá que, al recibir una nueva petición, el *edge service* consulte con el *service registry* qué instancias hay disponibles para realizar el enrutado.

Ilustración 71: Anotaciones requeridas en el edge service

El siguiente paso ha sido añadir la configuración de Eureka y la tabla de enrutados, donde se han especificado diferentes rutas a las que irán dirigidas las peticiones y los identificadores del servicio al que deberían ser enrutadas.

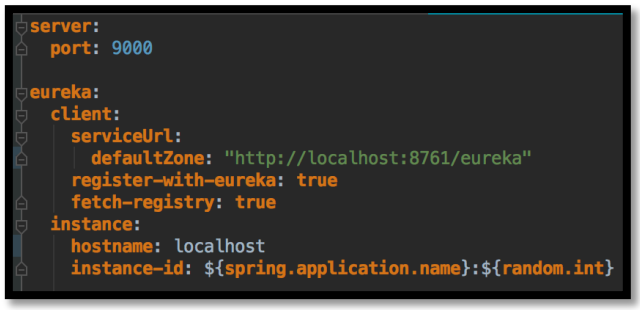


Ilustración 72: Configuración del edge service

Además de redirigir las peticiones al servicio correcto, *Zuul* también permite realizar balanceo de carga entre las diferentes instancias haciendo uso de diferentes algoritmos mediante el la integración con Ribbon, cuyo funcionamiento se resume en el apartado 3.3. Para activar esta funcionalidad se ha añadido un parámetro de configuración, visible al final de la siguiente ilustración.

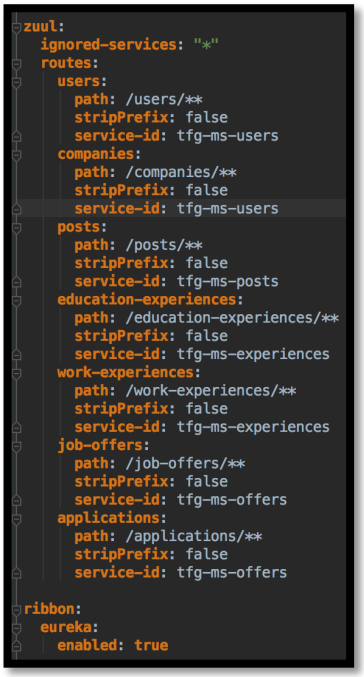


Ilustración 73: Rutas definidas en el edge service

9.2.3 Comunicación entre servicios

La comunicación entre servicios ha sido uno de los grandes retos del desarrollo del sistema de microservicios.

Estos pueden comunicarse de dos formas diferentes, síncrona y asíncronamente. Por ello, ha sido necesario implementar dos canales distintos de comunicación utilizando diversas tecnologías.

A continuación se explican los detalles de cada una de ellas así como el proceso de implementación.

Peticiones HTTP

En algunos casos, un servicio debía comunicarse con otro y esperar a la respuesta de este para decidir cómo actuar. Este es un ejemplo de comunicación síncrona y la solución elegida para implementarla han sido las peticiones HTTP.

Dichas peticiones se han realizado con *RestTemplate*, una herramienta de *Spring* que permite realizar peticiones HTTP de forma sencilla especificando únicamente la dirección URL y el objeto que esperas obtener.

```
@Autowired
private RestTemplate restTemplate;

@Override
public Boolean userExists(Long userId) {
    return restTemplate.getForObject( url: "http://tfg-ms-users/users/" + userId + "/exists", Boolean.class);
}
```

Ilustración 74: Petición HTTP con RestTemplate

En la ilustración anterior se puede ver un ejemplo de la utilización de *RestTemplate* en el proyecto, haciendo uso también de el identificador del microservicio usado en *Eureka* en lugar de la IP. Esto permite que las instancias puedan cambiar con el tiempo y no sea necesario modificar las direcciones a las que se hace la petición.

Eventos

El mayor reto en cuanto a la comunicación entre servicios ha sido la comunicación asíncrona, utilizada para notificar al resto de servicios, por ejemplo, al eliminar un usuario, por si alguno disponía de publicaciones, experiencias o candidaturas vinculadas a este y quería actuar en consecuencia.

La solución utilizada para gestionar este tipo de comunicación han sido las colas de eventos. Su funcionamiento es sencillo, existe un bróker de mensajes, en el caso de este proyecto se ha optado por *RabbitMQ,* encargado de gestionar los mensajes que recibe y publicarlos en las correspondientes colas, de las que podrán ser consumidos por aquellos servicios interesados.

Para implementar esta funcionalidad, lo primero ha sido obtener un bróker de mensajes. Este se ha ejecutado directamente en un contenedor docker utilizando la imagen *rabbitmq:3.6.9-management,* disponible en *DockerHub.*

```
➜ ~ docker run -rm -d --hostname rabbitmq --name rabbitmq -p 15672:15672 -p 5672:5672 rabbitmq:3.6.9-management
```

Ilustración 75: Comando ejecución contenedor RabbitMQ

Una vez iniciado el contenedor, el panel de control ya se encuentra accesible en la ruta *localhost:15672,* por lo que accediendo a el se observan, entre otras cosas, los canales de los que dispone, entre los que se encuentra *tfg.users,* que será el que utilice el sistema y que se explicará a continuación.

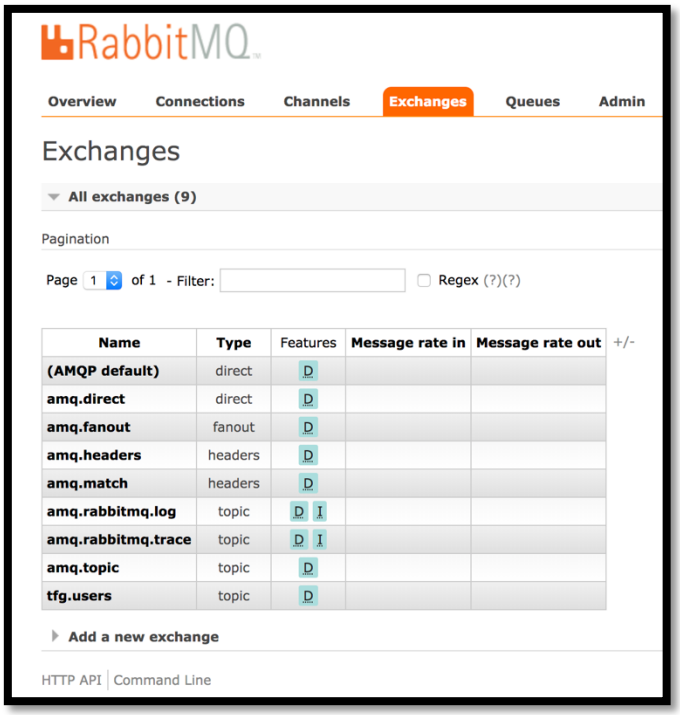


Ilustración 76: Panel de gestión de RabbitMQ

El siguiente paso ha sido configurar los servicios para integrarlos con el sistema de mensajes. Para ello lo primero ha sido añadir la configuración de *RabbitMQ* al fichero de configuración *application.yml.*

```yaml
spring:
  rabbitmq:
    host: localhost
    port: 5672
    username: guest
    password: guest
```

Ilustración 77: Parámetros de configuración de RabbitMQ

Por último, con tal de poder consumir y producir eventos, ha sido necesario añadir la anotación *@EnableBinding* y crear los consumers junto con la definición del canal.

Las siguientes ilustraciones muestra el consumer de el evento de borrado de usuarios en el microservicios de experiencias así como el canal del que se consumirán los eventos, que como se puede observar, coincide con el del canal de la Ilustración 76.

```
public interface UserChannel {
    String INPUT = "tfg.users";

    @Input(UserChannel.INPUT)
    SubscribableChannel input();
}
```

Ilustración 78: Canal de usuarios en el microservicio de experiencias

```
@Slf4j
public class UserMessageConsumer {

    @Autowired
    private UserMessageAdapter adapter;

    @StreamListener(UserChannel.INPUT)
    private void handle(CustomMessage message) {
        log.info("UserMessageConsumer consumed message:" + message);
        switch (message.getAction()){
            case DELETE:
                adapter.userDeleted(Long.valueOf(message.getContent().toString()));
                break;
        }
    }
}
```

Ilustración 79: Consumer de eventos de usuario en el ms de experiencias

El resto de información acerca del proceso de creación y configuración de un servicio se encuentra explicada en el apartado 8.3.3, dentro del sub-apartado de *Spring Cloud Stream*.

9.2.4 Bibliotecas propias

Con el fin de estandarizar los objetos necesarios para consumir y producir eventos a lo largo de todos los servicios, se ha creado una biblioteca con la definición de los mismos. Esto mejora la cambiabilidad del sistema ya que en caso de realizar cualquier cambio de tecnología o de añadir nuevos campos a

los objetos que encapsulan los mensajes, como por ejemplo metadatos tales como el nombre del servicio que lo ha producido o a que hora se ha hecho, únicamente será necesario modificar el sistema en un único sitio.

Dicha biblioteca, llamada *tfg-lib-messaging* se ha creado utilizando *Spring* y ha sido publicada en *Github*, repositorio en el que también se han alojado el resto de servicios.

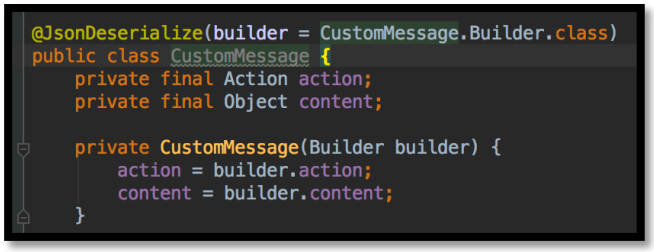


Ilustración 80: Wrapper de los mensajes enviados

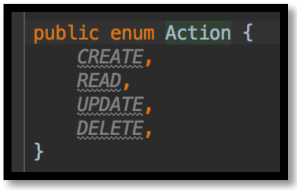


Ilustración 81: Acción que ha causado que el evento se produzca

Para poder hacer uso de la biblioteca en los diferentes servicios se ha utilizado de *JitPack,* que ha permitido importar como dependencia un proyecto publicado en un repositorio público de *GitHub*.

Ilustración 82: Uso de JitPack para importar dependencia desde Github

9.3 Realización de pruebas

Con el objetivo de verificar el correcto funcionamiento del sistema así como de generar confianza para facilitar futuros cambios en el mismo, se han creado tests encargados de probar las diferentes funcionalidades.

A continuación se explica en detalle el funcionamiento de los diferentes tipos de pruebas realizadas así como el objetivo de cada uno de ellos.

9.3.1 Tests unitarios

Los tests unitarios son los encargados de probar de forma aislada e independiente los diferentes componentes del sistema. El objetivo principal de este tipo de tests es el de dar información de forma rápida al desarrollador acerca del funcionamiento del mismo, la cual permitirá detectar posibles errores introducidos al realizar un cambio o fallos en la funcionalidad implementada.

Debido al tipo de proyecto, donde la mayoría de los casos de uso consisten en operaciones de creación, consulta, actualización y borrado de elementos, no se ha considerado necesario probar muchas de las clases, ya que la gran mayoría únicamente realizaban una llamada a otra clase y como mucho un mapeo de objetos de dominio a *DTO*, que son los objetos utilizados en los controladores.

Sí que se ha considerado oportuno probar los objetos de dominio, en los que se ha verificado que implementaban los métodos *equals*, *hash* code, así como los *getters*. Además, también se ha probado la lógica de validación de los mismos a la hora de crear una nueva instancia.

Ilustración 83: Ejemplo test unitario

En la ilustración anterior se puede observar los test unitarios encargados de probar la clase *Application.* Estos incluyen la anotación *@Test* de *JUnit,* que permitirá detectar qué métodos son test y ejecutarlos cuando sea necesario.

Para verificar de forma sencilla la correcta implementación de los métodos listados anteriormente, se ha utilizado *assertPojoMethodsFor,* de la biblioteca *pojo-tester.*

También se ha utilizado la opción *expected*, para verificar que al intentar crear una instancia del objeto de dominio con datos inválidos se lanzaba una excepción.

9.3.2 Tests de integración

Al contrario de los test unitarios , donde se prueba cada componente de forma aislada, los tests de integración permiten probar el sistema completo para verificar que los diferentes componentes funcionan bien unos con otros.

Este tipo de tests prueban desde los controladores, simulando peticiones *HTTP,* hasta la base de datos, donde realizan una comprobación de los cambios realizados.

La Ilustración 84 muestra un ejemplo de test de integración que prueba que al realizar una petición al controlador de usuarios para actualizar uno de ellos, esta actualización se realiza correctamente.

Para ello se simula, mediante la biblioteca *RestAssured*, una petición *PUT* con un objeto de usuario en el body cuyo nombre ha sido modificado y más

tarde se verifica el código de la respuesta, que el formato de esta es *JSON* y que el objeto devuelto contiene el nombre modificado.

Además de esta prueba, también se verifica el estado inicial y final de la base de datos con las anotaciones *@DatabaseSetup* y *@ExpectedDatabase* de *DBUnit* para comprobar que los datos del usuario se han actualizado correctamente.

```
@Test
@DatabaseSetup("shouldUpdateUser_whenUpdateUser.xml")
@ExpectedDatabase(
        value = "shouldUpdateUser_whenUpdateUser_assert.xml",
        table = "USERS",
        assertionMode = NON_STRICT
)
public void shouldUpdateUser_whenUpdateUser() {
    given() RequestSpecification
            .body(getUser().withName("NAME_MOD").build()) RequestSpecification
            .when() RequestSpecification
            .put( path: "/users") Response
            .then() ValidatableResponse
            .body( path: "name", is( value: "NAME_MOD")) ValidatableResponse
            .contentType(ContentType.JSON) ValidatableResponse
            .statusCode(HttpStatus.OK.value());
}
```

Ilustración 84: Test de integración de ejemplo

La siguiente ilustración muestra contenido de uno de los archivos utilizados en los tests de integración para especificar el estado inicial de la base de datos.

```
<dataset>
    <USERS ID="1" />
    <COMPANIES ID="1" />
    <JOB_OFFERS ID="1" COMPANY_ID="1" />
    <EDUCATION_EXPERIENCES ID="1" USER_ID="1" />
    <WORK_EXPERIENCES ID="2" USER_ID="1" />
    <APPLICATIONS ID="1" USER_ID="1" OFFER_ID="1" />
</dataset>
```

Ilustración 85: Contenido de la base de datos para un test de integración

Uno de los inconvenientes de los tests de integración es que, al probar todo el flujo, llevaría más tiempo determinar el punto exacto en el que el sistema ha fallado en caso de ocurrir un error.

10 Puesta en producción

Este apartado incluye toda la información referente al proceso de puesta en producción tanto del sistema monolítico como del sistema basado en microservicios así como a la configuración de la infraestructura para su correcto funcionamiento.

10.1 Sistema monolítico

Antes de comenzar, el primer paso ha sido el de decidir qué plataforma se utilizará. En el caso de este proyecto, tanto el sistema monolítico como el sistema basado en microservicios se han desplegado en *AWS,* una plataforma de *Amazon* que ofrece servicios de computación en la nube.

El primer paso para poder poner en producción el sistema monolítico ha sido generar un archivo *JAR.* Para ello se ha ejecutado el comando *./gradlew bootJar* en el directorio del proyecto, lo cual genera un archivo en el directorio */build/libs.*

De forma opcional se puede añadir al archivo de configuración *build.gradle* información sobre el JAR a generar, como el nombre o la versión, por ejemplo.

```
bootJar{
    baseName = 'tfg-monolith'
    version = '1.0'
}
```

Ilustración 86: Configuración del archivo JAR

Una vez generado el ejecutable ha abierto el servicio *Elastic Beanstalk* de *AWS* y creado una nueva aplicación.

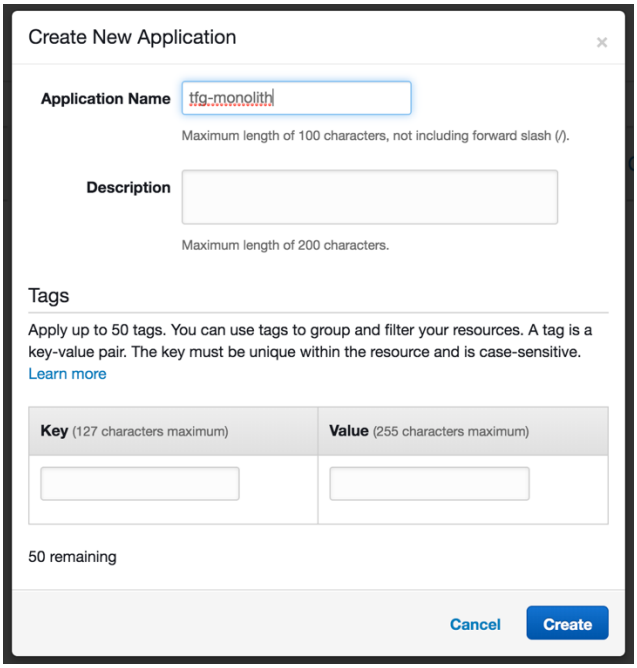


Ilustración 87: Create new Elastic Beanstalk application

Elastic Beanstalk es un servicio para desplegar y escalar aplicaciones web y servicios en la nube que se encarga automáticamente del auto escalado, balanceo de carga y salud de los servicios, entre otras funcionalidades.

Una vez creada la aplicación, es necesario crear un nuevo entorno dentro de la misma, para ello se ha seleccionado la opción *Web Server environment,* lo que abre una pantalla donde realizar la configuración base del entorno.

Aquí será se ha elegido *Java* como plataforma para el sistema y cargado el archivo JAR generado anteriormente.

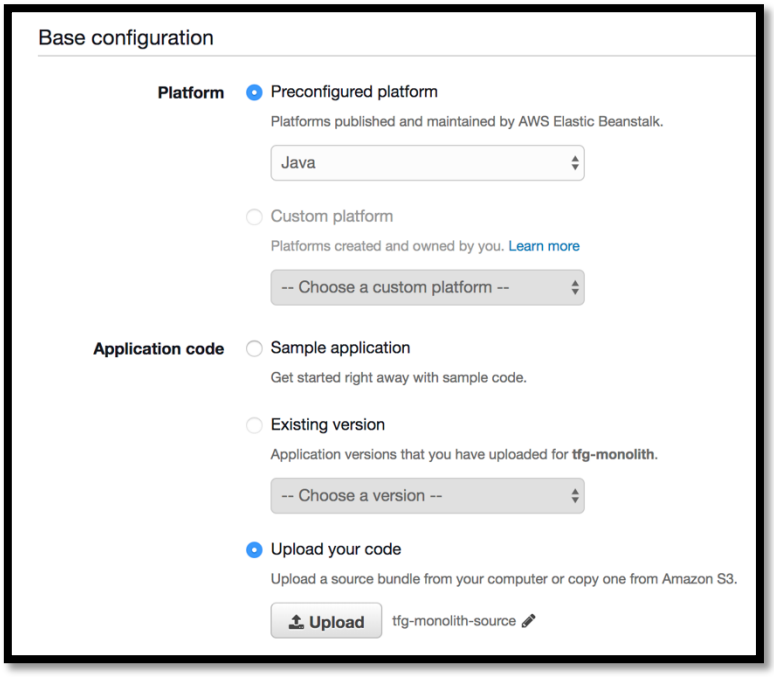


Ilustración 88: Configuración base del entorno de Elastic beanstalk

Una vez hecho esto y antes de crear el entorno, se han modificado también algunos parámetros como el auto escalado, estableciendo el número máximo de instancias a 2, y el tipo de instancia, que se ha cambiado de *t2.micro* a *t2.medium*.

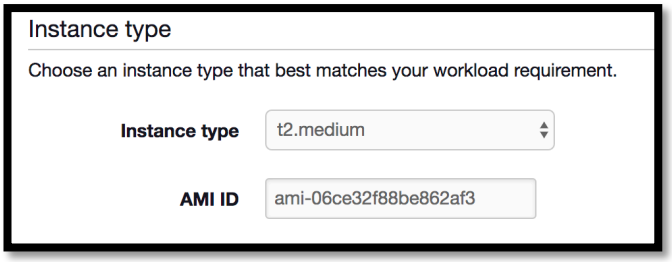


Ilustración 89: Configuración instancias Elastic beanstalk

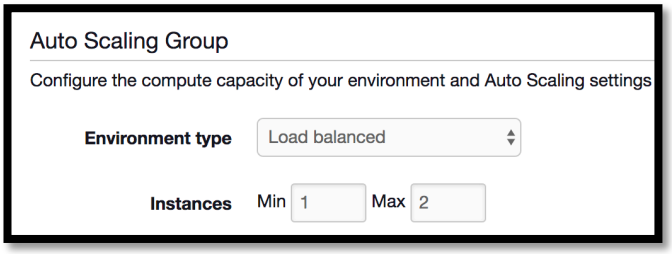
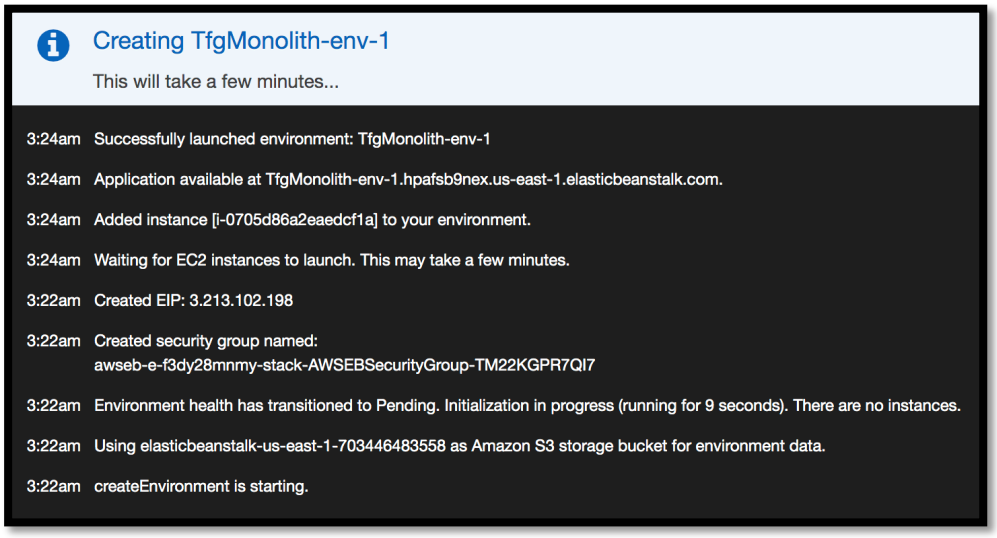
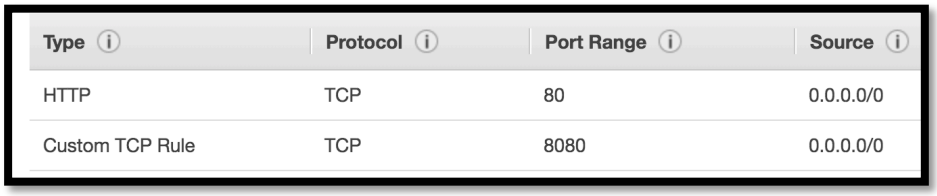


Ilustración 90: Configuración de auto-scaling Elastic beanstalk

Al finalizar la configuración, *Elastic Beanstalk* comienza a crear todo lo necesario para realizar el despliegue, como un contenedor S3 para almacenar los logs, un nuevo security group, la nueva IP y las instancias EC2 necesarias.



Una vez terminado el proceso de creación se ha accedido a la dirección DNS generada,obteniendo un error debido a un problema de permisos. Para solucionarlo se ha abierto el servicio VPC y modificado el *security group* creado para permitir peticiones en el puerto 8080, que es el utilizado por el sistema.



Type	Protocol	Port Range	Source
HTTP	TCP	80	0.0.0.0/0
Custom TCP Rule	TCP	8080	0.0.0.0/0

Ilustración 91: Configuración security group

Con este último paso finaliza el proceso de despliegue del sistema monolítico, pudiendo acceder sin problemas a través de la dirección DNS.

10.2 Sistema basado en microservicios

Desplegar en la nube el sistema basado en microservicios requiere de una mayor infraestructura que el sistema monolítico, así como de una correcta configuración de la red con tal de permitir la comunicación entre servicios, lo que aumenta la complejidad.

En la siguiente imagen, se puede observar la toda la infraestructura necesaria.

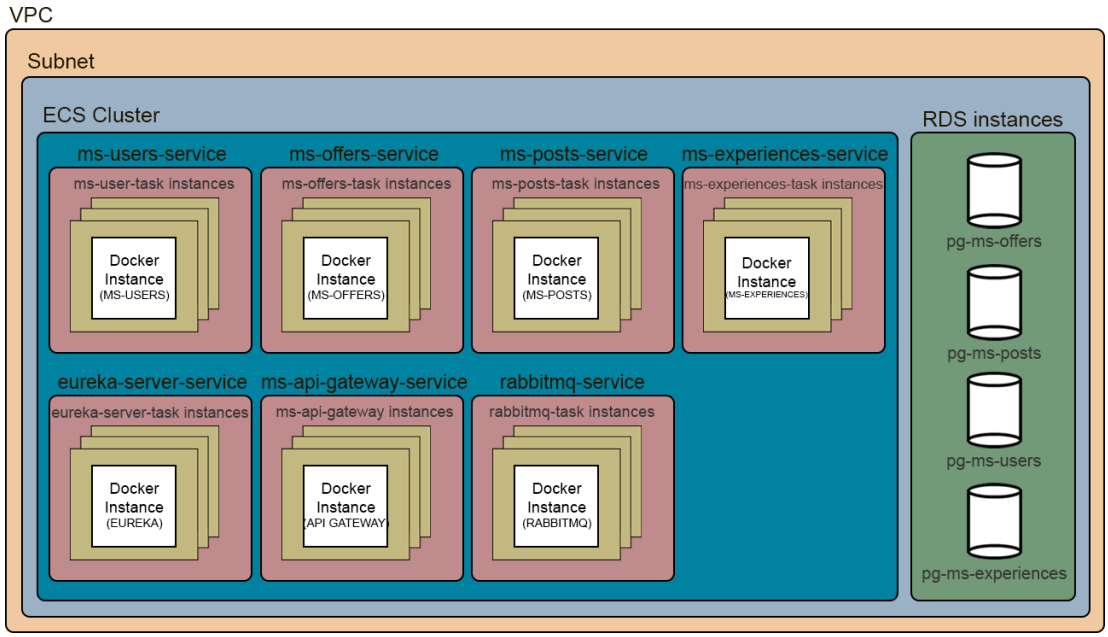


Ilustración 92: Infraestructura AWS sistema de microservicios

10.2.1 Configuración de la red

Lo primero a realizar es la creación de una red privada mediante el servicio *VPC,* tal y como se muestra en la Ilustración 93 y acto seguido sasociarla a

un *Internet Gateway*, que actúa como router virtual y permite el acceso a dicha red desde Internet.

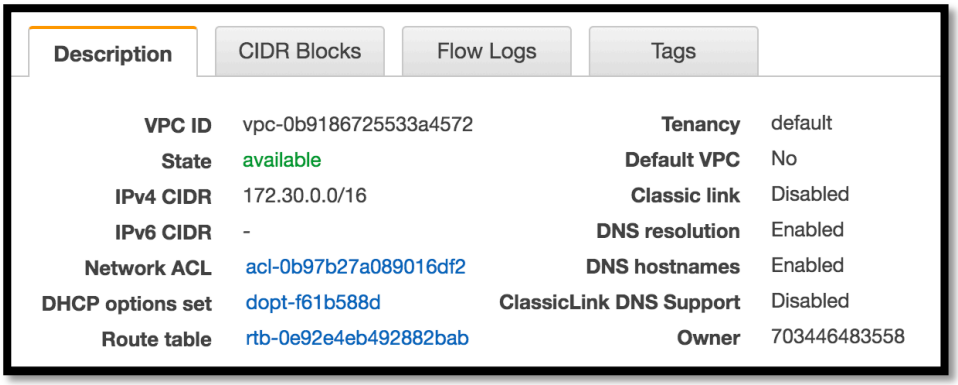


Ilustración 93: Configuración de la VPC

Una vez creada la red, se han creado también algunas subredes, aunque finalmente se ha acabado utilizando únicamente una de ellas, la 172.30.3.0/24. En la siguiente imagen se muestra la configuración realizada.

Subnet ID	State	VPC	IPv4 CIDR	Available IPv4	Availability Zone
subnet-01fa0ed75a9e80c27	available	vpc-0b9186725533a4572 ...	172.30.2.0/24	249	us-east-1c
subnet-05d461e470b8f7cfb	available	vpc-0b9186725533a4572 ...	172.30.4.0/24	251	us-east-1e
subnet-097b4e4b50ff060c1	available	vpc-0b9186725533a4572 ...	172.30.3.0/24	250	us-east-1d
subnet-0a926394bf2c2d081	available	vpc-0b9186725533a4572 ...	172.30.5.0/24	249	us-east-1f
subnet-0de596fc5b2b64362	available	vpc-0b9186725533a4572 ...	172.30.1.0/24	250	us-east-1b
subnet-0faadba5d6f9e475b	available	vpc-0b9186725533a4572 ...	172.30.0.0/24	251	us-east-1a

Ilustración 94: Configuración subredes AWS

10.2.2 Creación de registros ECR

El siguiente paso consiste en crear repositorios que proporcionen acceso a las diferentes versiones de las imágenes *Docker* de los servicios.

Para ello, se ha hecho uso del servicio *Elastic Container Registry* o *ECR,* que es un registro de contenedores que permite almacenarlos, gestionarlos y desplegarlos de forma sencilla.

Primero se han creado los repositorios necesarios, a los que se han subido sus imágenes *Docker* correspondientes mediante la *Command Line Interface* de *AWS.*

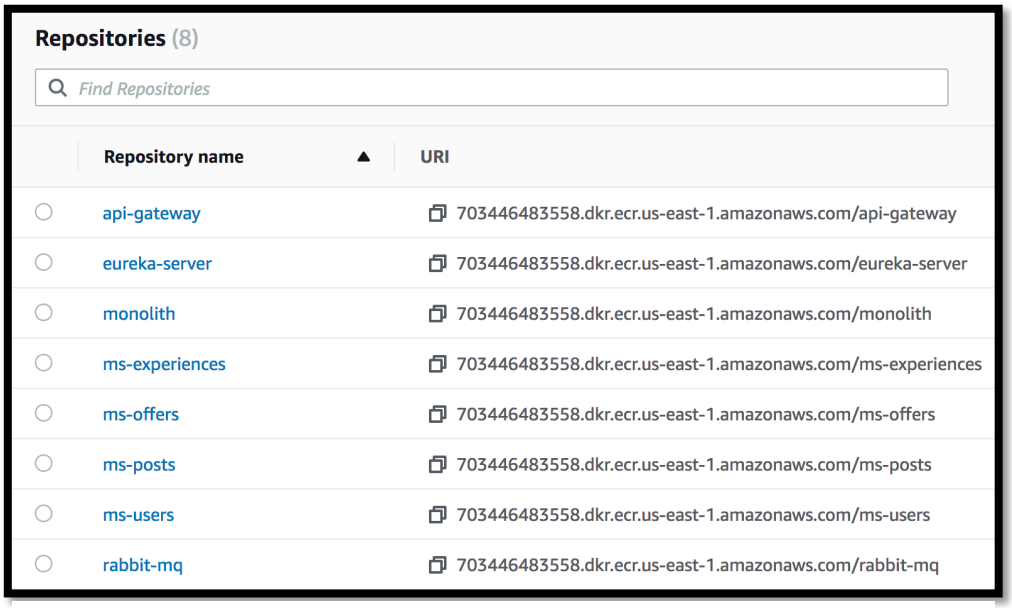


Ilustración 95: Listado de repositorios ECR

10.2.3 Creación de las bases de datos

Para alojar las instancias de las bases de datos se ha hecho uso del servicio de bases de datos relacionales que Amazon ofrece, llamado RDS, que permite definir, operar, escalar bases de datos relacionales en la nube.

Para su creación se ha elegido el sistema PostgreSQL, sin replicación en diferentes regiones y en instancias *db.t2.medium,* que cuentan con 2 vCPU y 4GiB de memoria. Se ha utilizado la red y subred previamente creadas aunque, con el fin de hacerla accesible directamente desde mi ordenador, se le ha asignado una dirección DNS accesible públicamente.

En la siguiente ilustración se muestran las diferentes instancias creadas al finalizar el proceso.

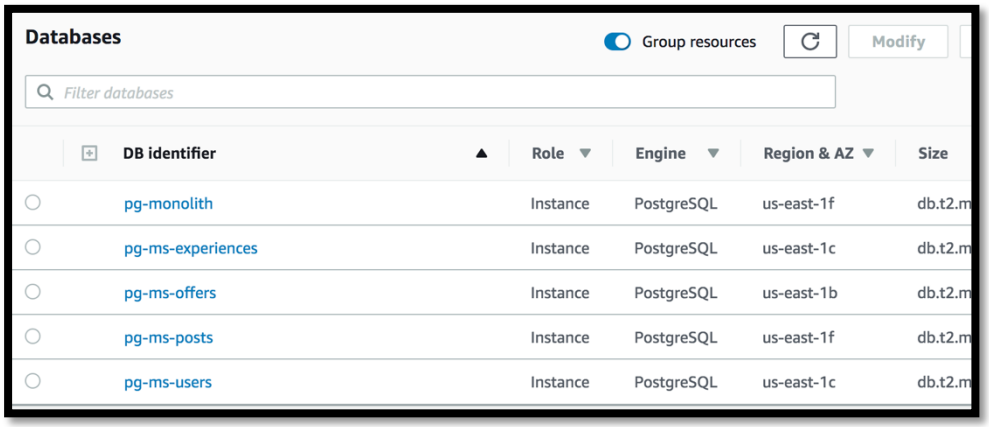


Ilustración 96: Listado de instáncias RDS

10.2.4 Creación de definiciones de tareas

El último preparativo necesario antes de poder realizar la creación del clúster consiste en definir de tareas.

Una tarea permite especificar qué contenedores se utilizarán, que recursos necesitarán, si estarán enlazados o que puertos usarán.

Dos contenedores deberían ir dentro de la misma tarea si tienen el mismo ciclo de vida, deben compartir recursos o necesitan estar dentro del mismo host. Dado que uno de los objetivos al usar microservicios es mejorar la escalabilidad, cada uno de ellos se ha introducido en una tarea diferente, lo que permite que estos puedan ser escalados sin necesidad de levantar instancias del resto de servicios.

La siguiente ilustración muestra la configuración de un contenedor dentro de la definición de una tarea. Se puede observar la imagen utilizada, la cual se consulta de los repositorios ECR previamente creados, el puerto expuesto así como variables de entorno que el servicio utilizará para configurar las conexiones.

Esto último, aunque no es necesario, mejora la seguridad ya que evita guardar las contraseñas y direcciones ip dentro del código, que más tarde se alojará en un repositorio público.

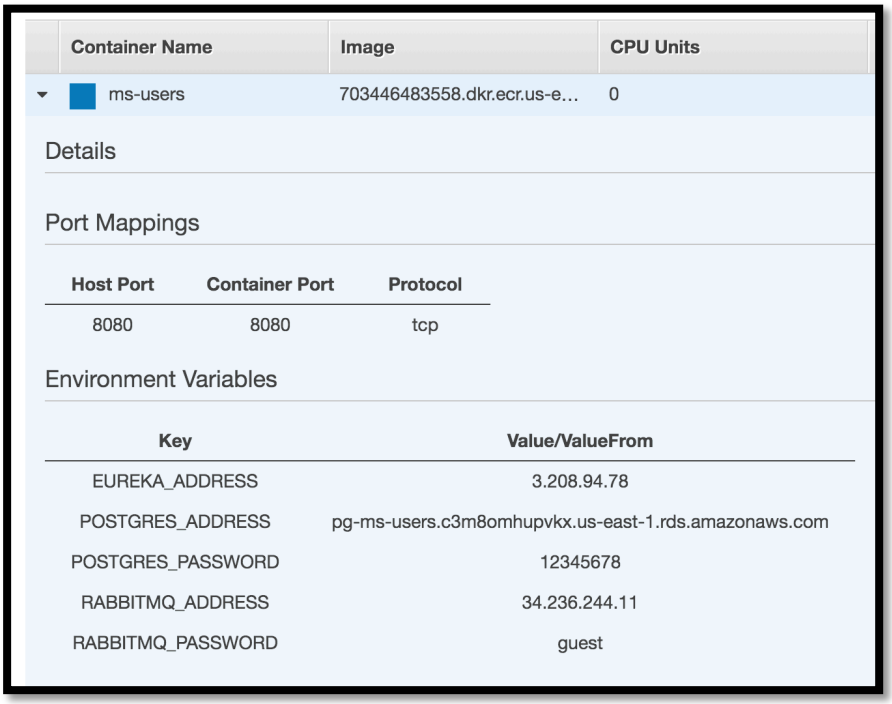
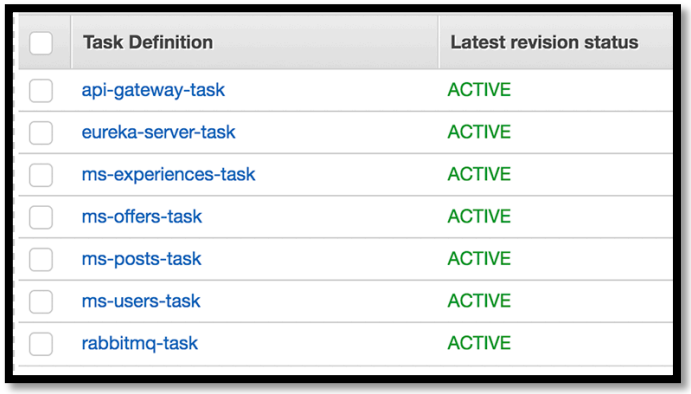


Ilustración 97: Configuración contenedor en definición de tarea

En la siguiente ilustración se puede observar el estado al finalizar el proceso de creación de todas las definiciones de tareas.



Task Definition	Latest revision status
api-gateway-task	ACTIVE
eureka-server-task	ACTIVE
ms-experiences-task	ACTIVE
ms-offers-task	ACTIVE
ms-posts-task	ACTIVE
ms-users-task	ACTIVE
rabbitmq-task	ACTIVE

Ilustración 98: Listado de definiciones de tarea

10.2.5 Creación del clúster

Un clúster ECS consiste en una agrupación lógica de tareas o servicios. Para su creación únicamente ha sido necesario especificar el nombre, ya que son los servicios que contiene los que incluirán toda la configuración.

Un servicio permite especificar cuántas copias de una definición de tarea se ejecutarán en el clúster. Opcionalmente, estos permiten la configuración de un balanceador de carga ELB, aunque esto no ha sido necesario ya que el balanceo de carga del sistema se ha realizado mediante *Ribbon* en el servicio *api-gateway*.

Para crear los servicios, ha sido necesario necesario especificar la definición de tarea, el número instancias de dicha tarea que debían ejecutarse, la red y subred así como las políticas de auto escalado, entre otros parámetros.

Como se puede observar en la Ilustración 99, en el caso de los microservicios, se ha definido una política de auto escalado al sobrepasar el 80% de utilización de la CPU media entre las instancias disponibles, definiendo un mínimo de 1 instancia, un máximo de 4 y un número deseado de 2.

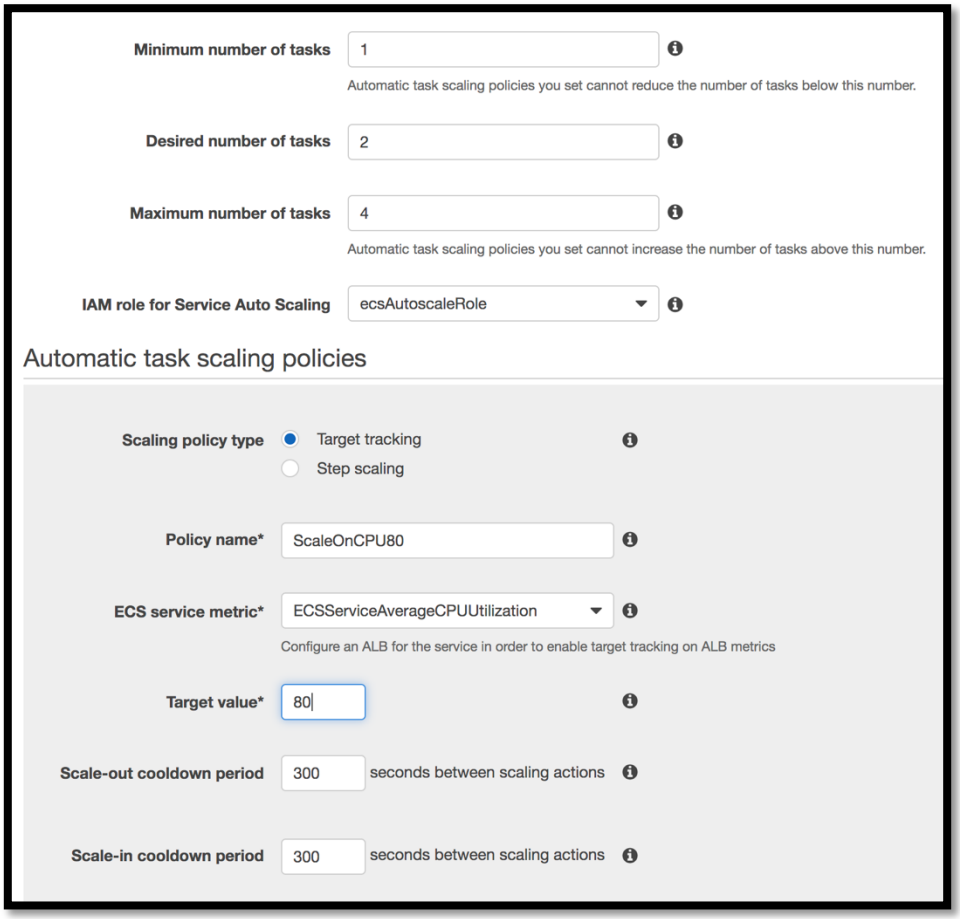


Ilustración 99: Política de auto escalado de microservicios

En la siguiente ilustración se pueden ver los diferentes servicios definidos en el clúster junto con información acerca de ellos, como por ejemplo, la definición de tarea que utiliza y el número de instancias ejecutándose.

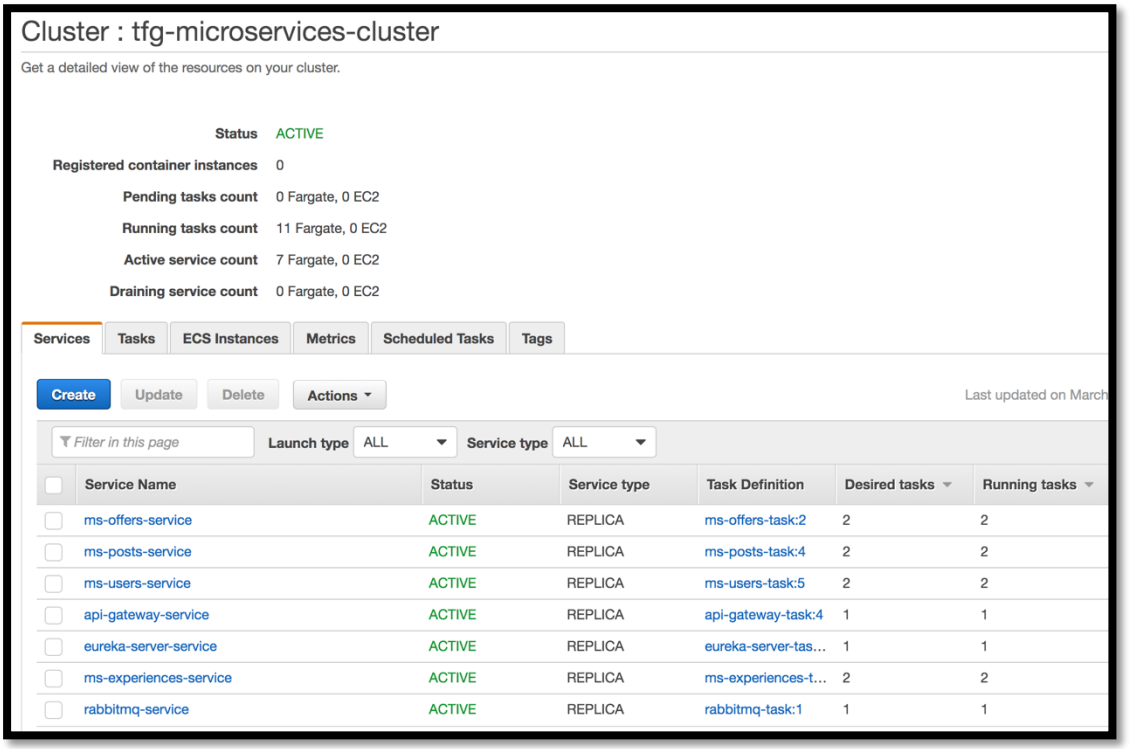


Ilustración 100: Información del clúster ECS

11 Análisis del rendimiento

Para la realización de las pruebas de rendimiento se ha utilizado *Jmeter,* un software *open-source* desarrollado en Java diseñado para realizar todo tipo de pruebas de rendimiento.

Con el objetivo de poder determinar si el rendimiento de ambos sistemas era suficientemente bueno, se definieron una serie de criterios:

- El número mínimo de request por segundo que el sistema debe ser capaz de gestionar es de 3000.
- El tiempo de respuesta máximo permitido se establece en 300ms, dado que a partir de este la experiencia de usuario se degradaría [24].

A continuación se explican los pasos seguidos para realizar las pruebas así como así como los resultados obtenidos.

11.1 Sistema monolítico

11.1.1 Creación de la prueba

El primer paso para realizar las pruebas de rendimiento del sistema monolítico ha sido crear un test de carga y definir un grupo de *threads* (que actuarán como usuarios).

Para ello se definen 375 usuarios, que realizando 8 peticiones por segundo cada uno suman un total de 3000 peticiones por segundo. Para evitar picos en la gráfica, se define un tiempo de creación de 60 segundos durante el cual se irán creando *threads* de forma gradual.

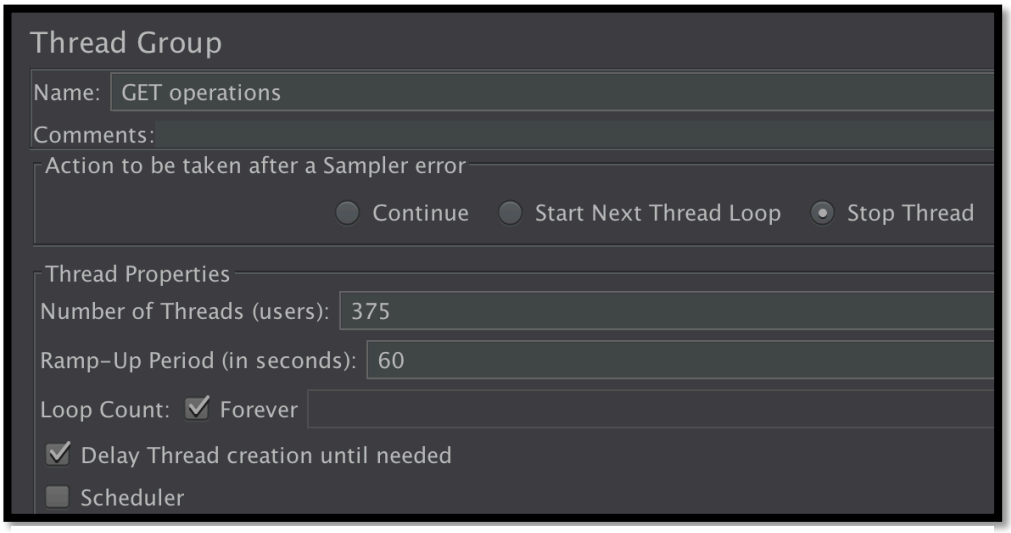


Ilustración 101: Creación threads JMeter

Una vez creado el grupo de *threads,* es necesario añadir las peticiones HTTP que deben realizar. En la siguiente ilustración se muestra una de las peticiones creadas.

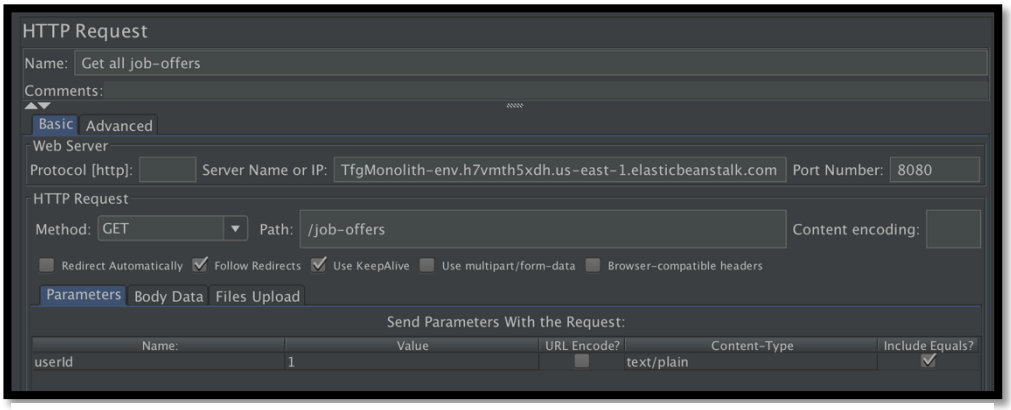


Ilustración 102: Ejemplo petición HTTP JMeter

Por último, será necesario añadir diferentes *listeners,* que permitirán recoger los resultados de las peticiones y mostrarlos de diversas formas. Algunos listeners útiles son *Response Time Graph*, *Aggregate Graph* y *View Results Tree*. A continuación se muestra la estructura final de la prueba antes de se ejecución.

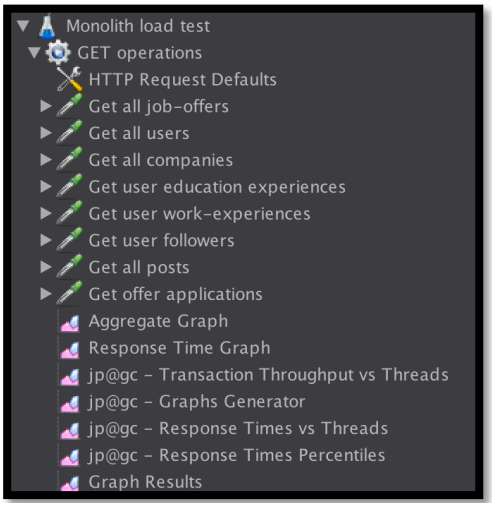


Ilustración 103: Estructura de la prueba de Jmeter (monolito)

11.1.2 Ejecución de la prueba

Una vez finalizado el proceso de creación de la prueba se podrá proceder a su ejecución. Para ello será necesario pulsar el botón start y *JMeter* se encargará de lanzar la prueba y mostrar los resultados en los diferentes *listeners* configurados.

Una vez se hayan recogido datos suficientes, en el caso de esta prueba fueron cuatro minutos, esta se podrá parar, destruyendo así todos los *threads* creados.

En la siguiente ilustración se puede observar la evolución del tiempo de respuesta. A partir de los datos obtenidos, el rendimiento parece correcto, ya que todas las peticiones realizadas a partir del primer minuto se encuentran entre 240 y 280 milisegundos.

También se observa que uno de los *endpoints* tarda aproximadamente el doble que el resto, llegando hasta los 520 milisegundos, por lo que se anota para revisarlo más adelante.

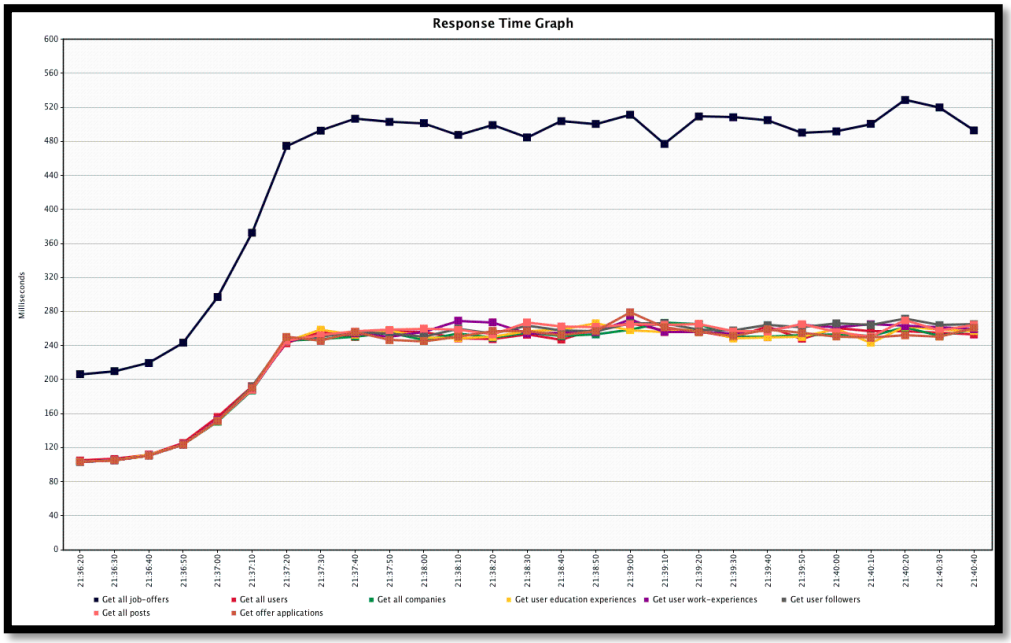


Ilustración 104: Gráfica del tiempo de respuesta (monolito)

La siguiente ilustración muestra que aproximadamente el 92% de las peticiones realizadas, excepto las de ofertas de empleo, se encuentran por debajo de los 300ms.

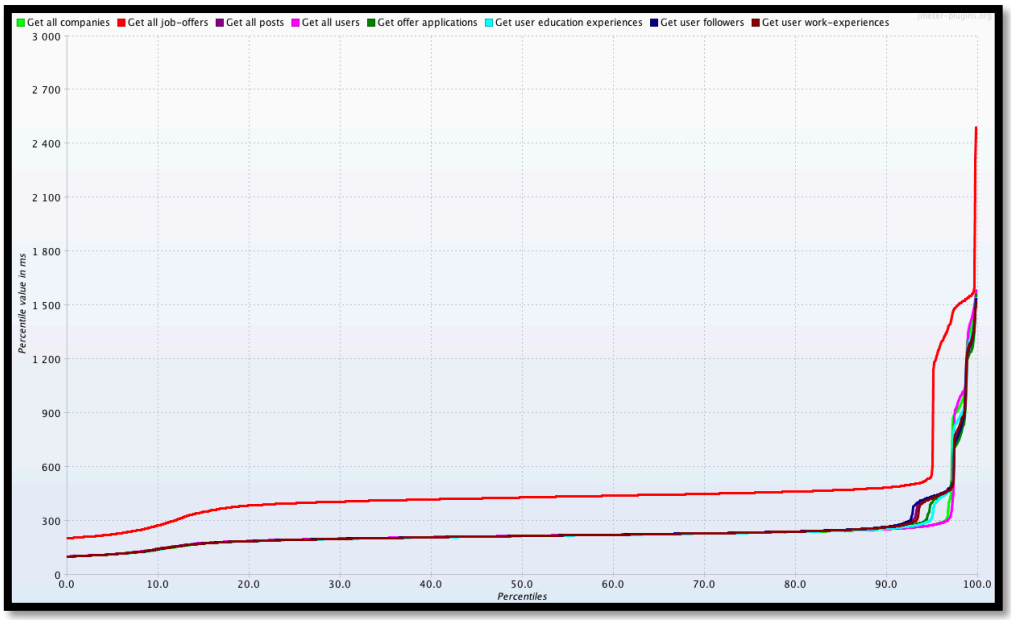


Ilustración 105: Gráfica percentiles de tiempo de respuesta

La siguiente ilustración muestra como evoluciona el tiempo de respuesta a medida que se incrementa el número de *threads*.

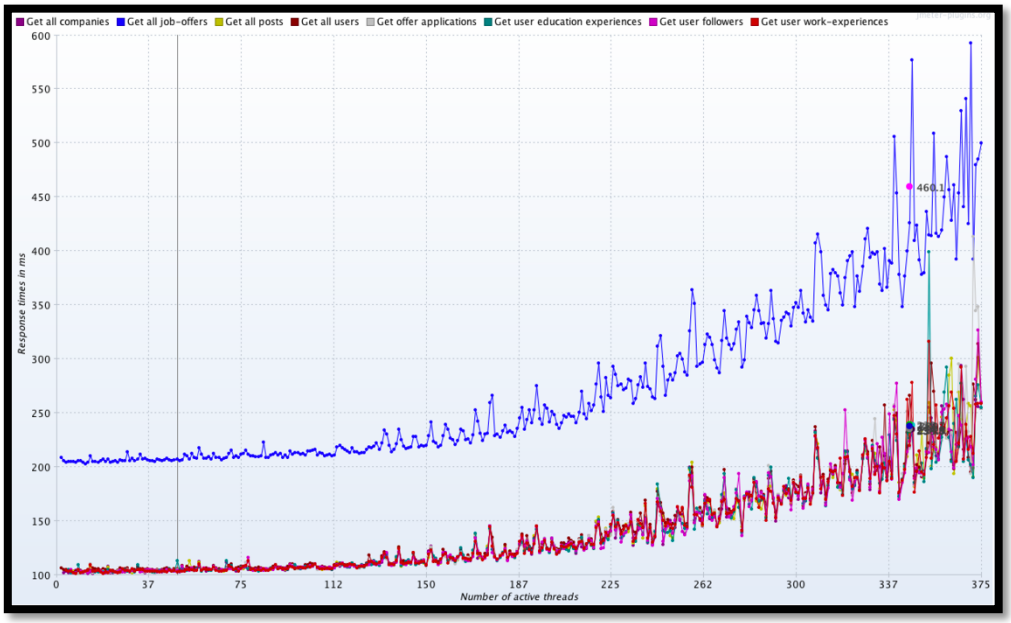


Ilustración 106: Tiempo de respuesta vs Threads (monolito)

Por último, se muestra toda la información recopilada sobre las respuestas obtenidas, desde el tamaño de la muestra para cada tipo de petición hasta la media, mediana y el porcentaje de errores.

Label	# Samples	Average	Median	90% Line	95% Line	99% Line	Error %
Get all job-offers	42590	459	431	486	559	1545	0.20%
Get all users	42505	234	217	255	276	1335	0.11%
Get all companies	42458	233	216	254	277	1310	0.10%
Get user education experiences	42416	234	216	257	311	1265	0.09%
Get user work-experiences	42377	238	216	265	426	1260	0.09%
Get user followers	42338	239	217	267	435	1249	0.11%
Get all posts	42292	238	216	266	432	1235	0.09%
Get offer applications	42252	234	216	260	400	1205	0.09%
TOTAL	339228	264	221	430	466	1329	0.11%

Ilustración 107: Resultados finales de la prueba (monolito)

11.1.3 Problemas encontrados

Teniendo en cuenta que se disponía de una política de auto escalado, se esperaba que a medida que el número de peticiones aumentase esta se disparase creando una nueva instancia y el tiempo de respuesta se redujese, permitiendo un mayor número de peticiones por segundo.

Sin embargo, durante la realización de las pruebas se ha comprobado que el servicio de auto escalado no ha llegado a dispararse, por lo que únicamente se ha utilizado una instancia.

Tras investigar la causa de este problema no se ha llegado a ninguna conclusión , ya que como se puede ver en la siguientes gráficas, la instancia EC2 tiene un consumo de aproximadamente un 50%, lo que es aceptable, así como un tráfico de entrada y salida correcto.

Ilustración 108: Métricas intáncia EC2 durante la prueba (monolito)

Dado que el problema no parece estar en la instancia EC2, se revisa también la instancia RDS, dónde se observa que los diferentes parámetros se encuentran dentro de la normalidad.

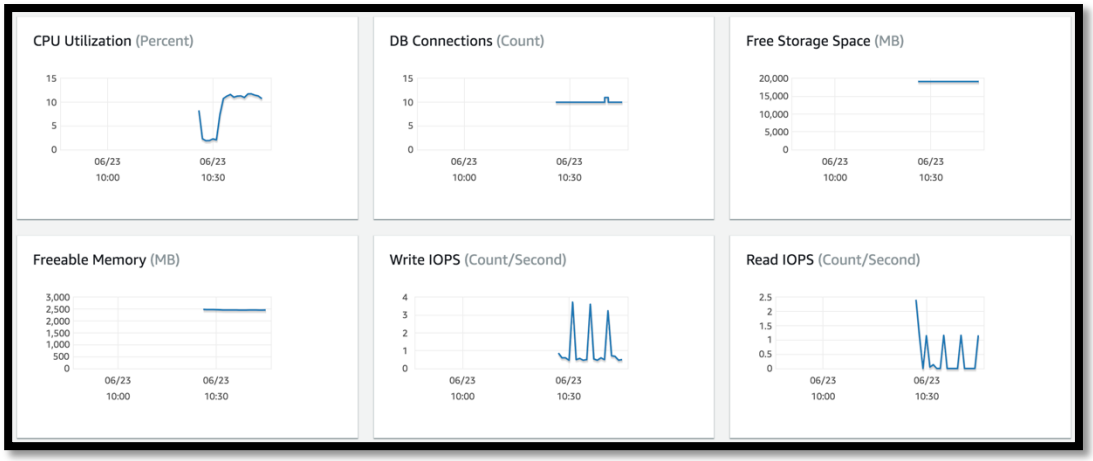


Ilustración 109: Métricas intáncia RDS durante la prueba (monolito)

Con el objetivo de encontrar el cuello de botella se incrementa tipo de instancia *EC2* de *t2.medium* a *t3.xlarge* y el tipo de instancia RDS de *t2.medium* a *t2.xlarge,* incrementando en ambos casos de 4Gb a 16Gb de memoria RAM y de 2vCPU a 4vCPU.

Al realizar la prueba tras el cambio, sin embargo, se puede observar que el consumo de CPU se ha reducido en ambos casos, como se muestra en las siguientes ilustraciones, pero el tiempo de respuesta se mantiene igual.

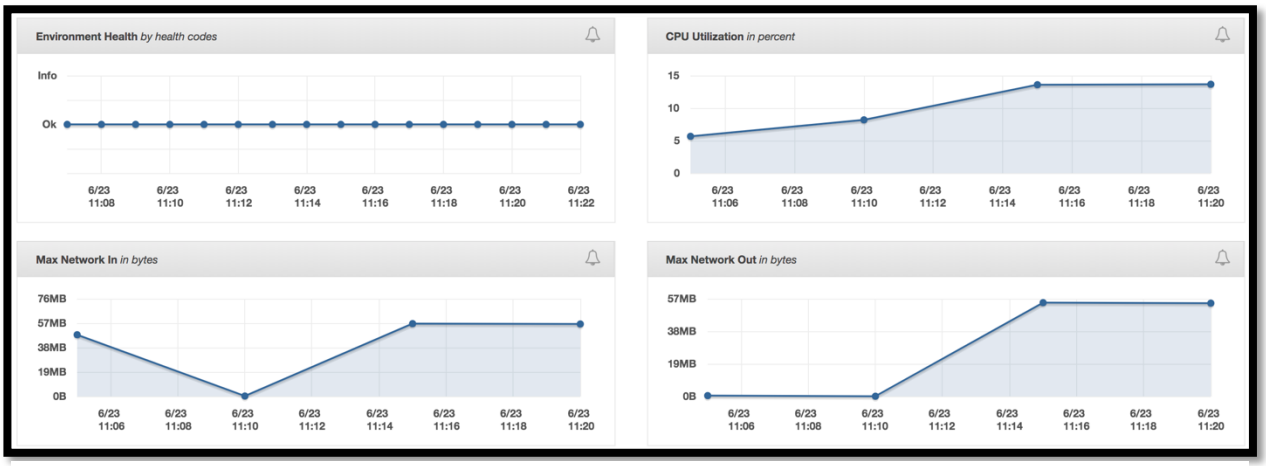


Ilustración 111: Métricas instáncia EC2 tras el cambio (monolito)

Ilustración 110: Métricas instáncia RDS tras el cambio (monolito)

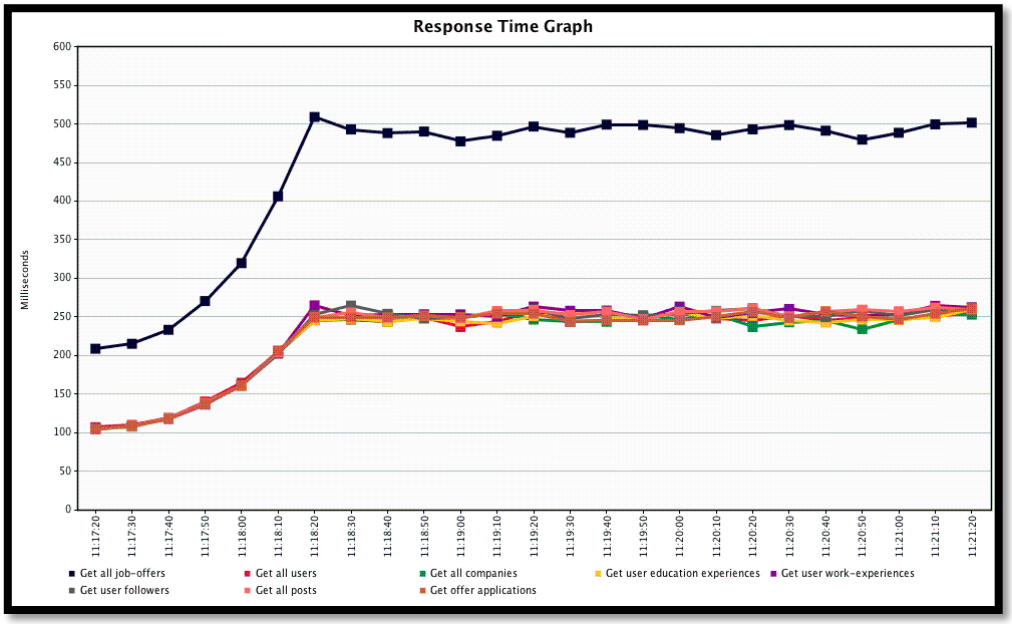


Ilustración 112: Tiempo de respuesta tras el cambio (monolito)

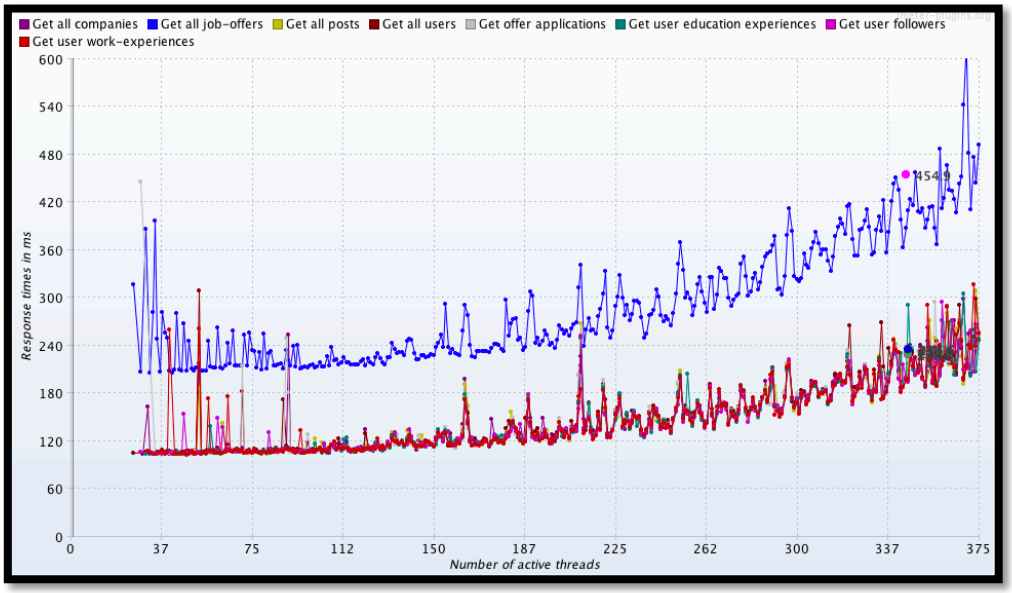


Ilustración 113: Tiempo de respuesta vs Threads tras el cambio (monolito)

Tras comprobar que el uso de CPU es correcto tanto en la instancia EC2 como en RDS, que el tráfico de entrada y salida es aceptable y que el número de conexiones máximas tanto de EC2 como de RDS se encuentran muy por encima de las utilizadas se desconoce el motivo por el que el tiempo de

respuesta se dispara al superar las 3000 peticiones, pudiendo llegar a los 800ms al aumentar el número de *threads* a 500.

11.2 Sistema basado en microservicios

11.2.1 Creación de la prueba

Para realizar las pruebas de rendimiento del sistema basado en microservicios se ha seguido el mismo proceso que en el sistema monolítico.

El primer paso ha sido crear el test de carga junto con el grupo de *threads,* que en este caso será también de 375 ejecutando cada uno 8 requests por segundo.

Una vez hecho esto, se definen igual que en el caso anterior todas las requests a realizar así como diferentes *listeners* para capturar los resultados y poder mostrarlos durante la ejecución de la prueba.

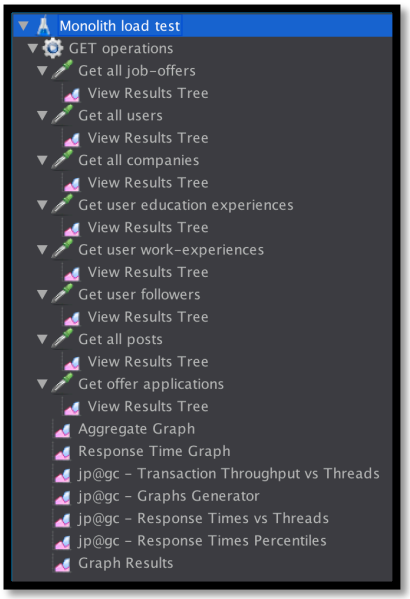


Ilustración 114: Estructura de la prueba de JMeter (microservicios)

11.2.2 Ejecución de la prueba

Una vez creada toda la estructura de la prueba se inicia la ejecución de la misma mediante el botón de iniciar de JMeter.

Como se puede observar en los primeros resultados obtenidos de algunos microservicios concretos son variados, llegando hasta los tres segundos, pero pronto se estabilizan.

Como se puede observar en las siguientes gráficas, el tiempo de respuesta de los microservicios es significativamente superior a la del monolito, estando en 600ms algunos endpoints y otros por encima del segundo.

En la Ilustración 116 también se puede observar como el numero de operaciones por segundo se encuentra por debajo de las 1000 en todos los servicios, llegando a las 400 en algunos de ellos.Ilustración 116: Transacciones por segundo vs Threads (microservicios)

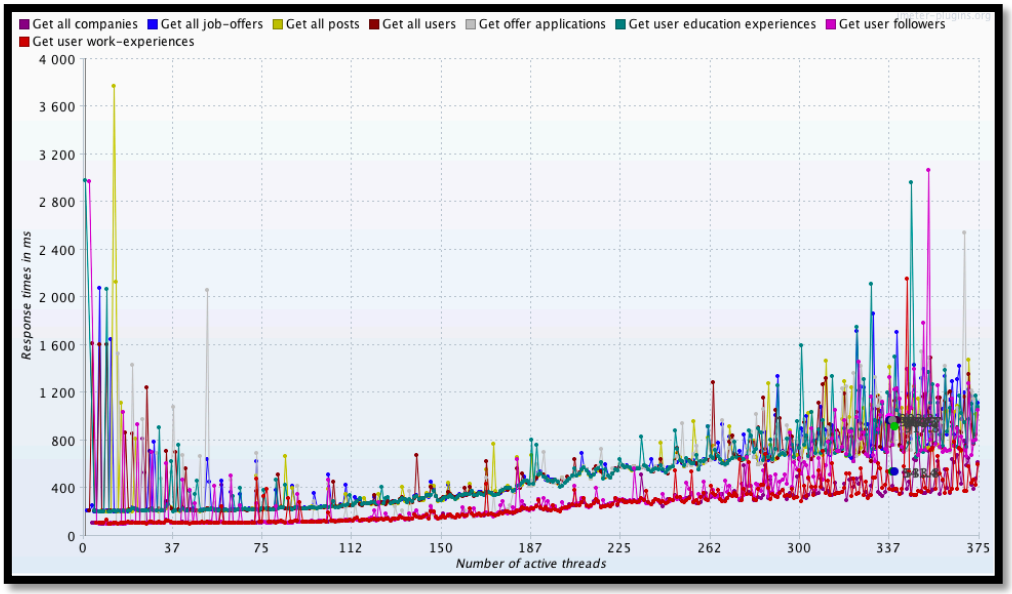


Ilustración 115: Tiempo de respuesta vs threads (microservicios)

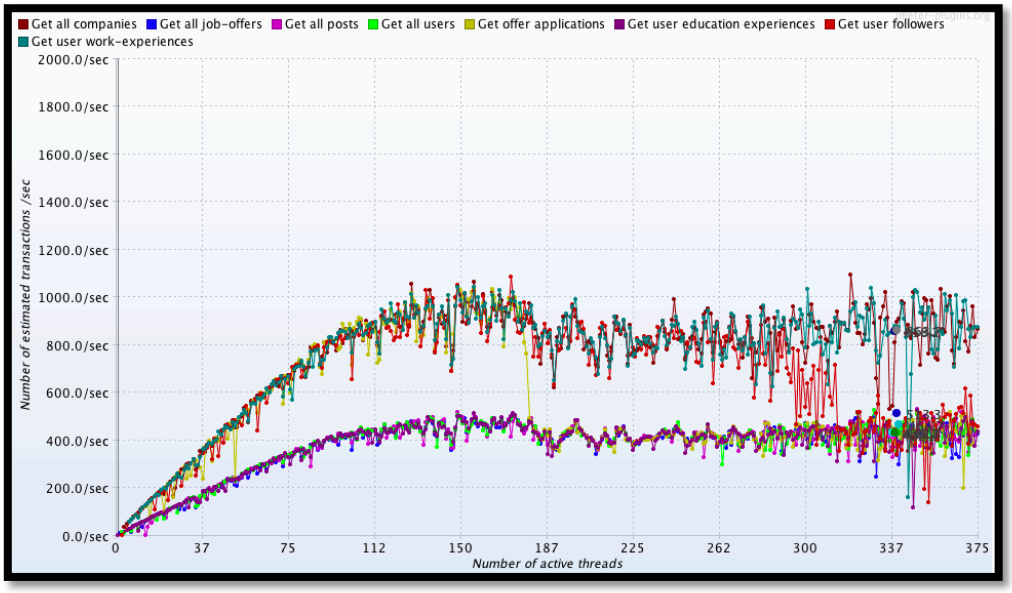


Ilustración 116: Transacciones por segundo vs Threads (microservicios)

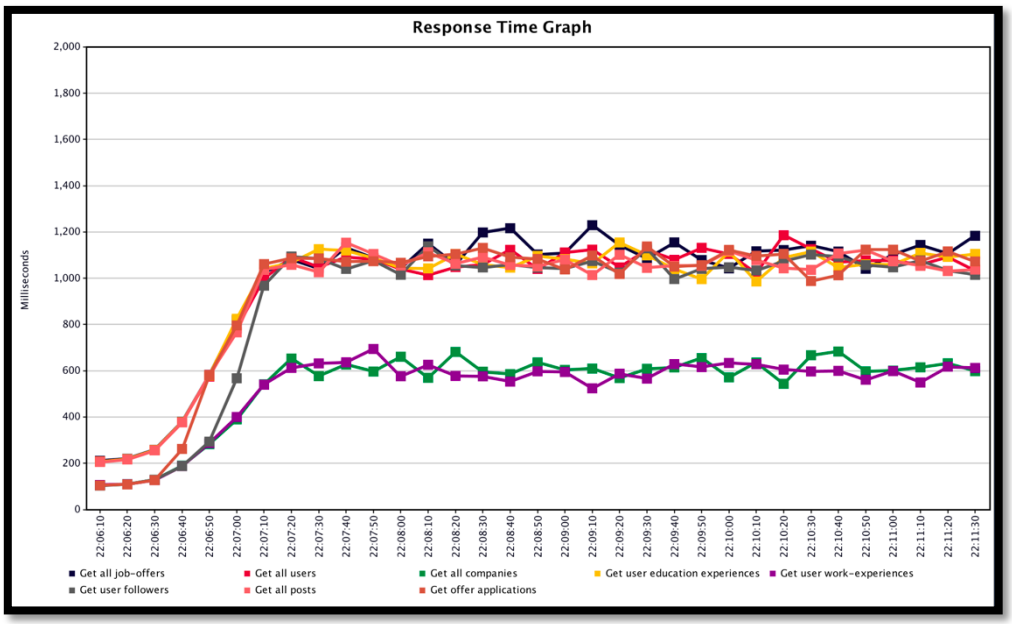


Ilustración 117: Gráfica del tiempo de respuesta (microservicios)

Label	# Samples	Average	Median	90% Line	95% Line	99% Line	Error %
Get all com...	16911	540	373	835	2143	2578	0.00%
Get user e...	16884	963	748	1845	2999	3398	0.00%
Get user w...	16831	533	372	815	1836	2585	0.00%
Get user fo...	16798	917	747	1839	3013	3386	0.00%
Get all posts	16741	961	747	1840	2982	3452	0.00%
Get offer a...	16694	963	750	1850	3024	3443	0.00%
TOTAL	134826	854	729	1825	2860	3344	0.00%

Ilustración 118: Resultados finales de la prueba (microservicios)

11.2.3 Problemas encontrados

El problema más evidente de los resultados de la prueba son los elevados tiempos de respuesta, sobretodo comparándolos con los obtenidos de la prueba de rendimiento del monolito.

Dado que cada servicio se ejecuta en una máquina de 4Gb de memoria RAM y 2vCPU, exactamente las mismas especificaciones que la máquina donde se ejecutaba el monolito, los resultados deberían ser mejores, o al menos parecidos.

Con el objetivo de determinar si esto es debido a las máquinas utilizadas, se modifica la tarea del microservicio de publicaciones para que pase a ejecutarse en una de 16Gb de RAM y 4vCPU. A continuación se muestran los resultados obtenidos, donde no se observa ningún cambio en el tiempo de respuesta a pesar de tener más recursos, por lo que se descarta este motivo.

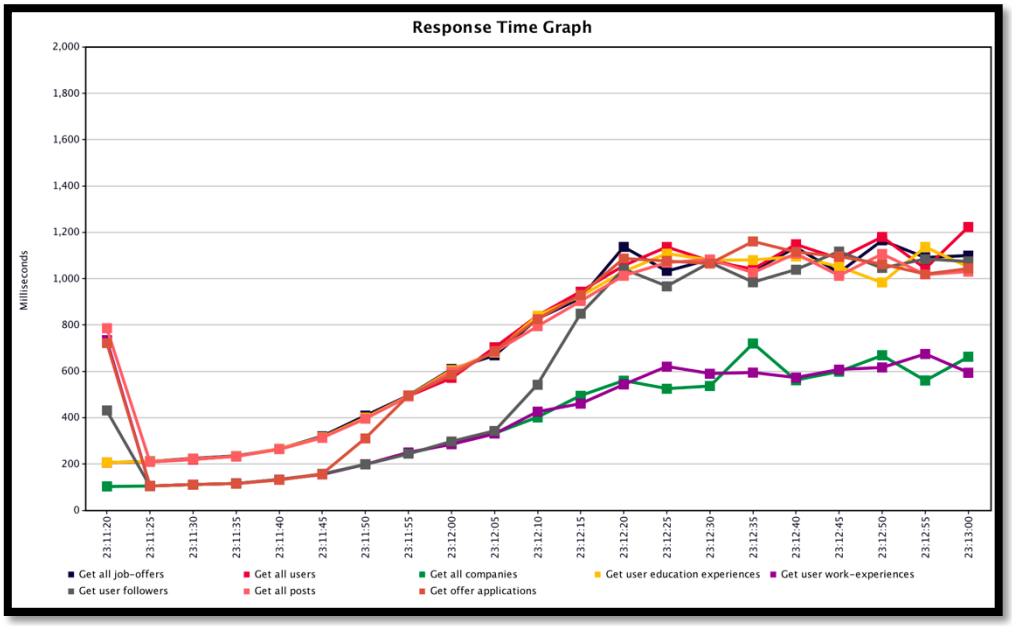


Ilustración 119: Tiempo de respuesta tras el cambio (microservicios)

A demás de los malos resultados obtenidos, se observa que, al igual que en la prueba de rendimiento del monolito, no se ha llegado a disparar el auto escalado de ninguno de los servicios.

El motivo, de nuevo, es que el uso de CPU durante la ejecución no ha llegado al umbral definido. A continuación se muestra el uso, tanto de CPU como de memoria, de los microservicios y de las instancias de base de datos. También se puede observar como el uso de CPU del microservicio de publicaciones es mas bajo que el del resto, debido al cambio realizado.

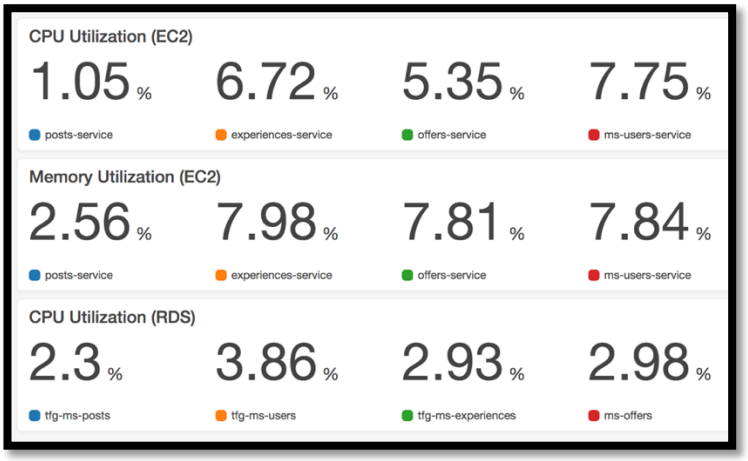


Ilustración 120: Uso de recursos de las instancias EC2 y RDS (microservicios)

Otro de los motivos que podrían estar afectando al resultado es que los microservicios se están ejecutando de *ECS*, dentro de contenedores *docker*, mientras que el monolito se ejecuta directamente en un servidor *Tomcat* utilizando la herramienta *Elastic Beanstalk*.

Aunque el uso de contenedores no debería afectar al tiempo de respuesta, pues estos son suficientemente ligeros como para no afectar al rendimiento del sistema, el servicio utilizado para ejecutarlos si podría hacerlo.

12 Planificación temporal

Uno de los primeros pasos a realizar durante la planificación del proyecto una vez definidas las tareas es la planificación temporal. Con ella se pretende validar la viabilidad del proyecto dado el alcance del mismo y el tiempo del que se dispone.

En esta sección se detalla la planificación temporal realizada en un primer momento, con una estimación del tiempo necesario para llevar a cabo cada una de las tareas, así como las desviaciones sufridas durante el transcurso del proyecto.

El inicio de este tuvo lugar el día 20 de febrero y su fecha de finalización prevista era el 19 de abril, aunque como se explicará más adelante esta fue modificada a una fecha posterior. La defensa del trabajo realizado debe llevarse a cabo una semana después de la fecha de finalización, por lo que el tiempo disponible hasta entonces se dedicará a la preparación de la defensa.

La duración total de este se estimó, por lo tanto, en 66 días, con una dedicación media diaria de entre 8 y 9 horas.

12.1 Definición de las tareas

Con el objetivo de mantener las tareas lo más organizadas posible, se ha dividido el proyecto en siete fases distintas.

12.1.1 Documentación inicial

La primera fase consiste en generar toda la documentación requerida para completar el hito inicial, la cual permite definir gran parte de los aspectos relacionados con la planificación del proyecto.

Se corresponde con la asignatura de Gestión de Proyectos, por lo que consta de un total de cuatro documentos que deben ser entregados para su posterior evaluación.

12.1.2 Desarrollo de los sistemas

Esta fase agrupa tanto el desarrollo del sistema basado en microservicios como el del sistema monolítico debido a la similitud de las tareas que contienen. Sin embargo, ambos desarrollos se tienen en cuenta como tareas separadas en la planificación.

Estudio previo

Con el fin de resolver cualquier duda que requiera ser abordada antes de comenzar el diseño y desarrollo del sistema, se ha decidido llevar a cabo un pequeño estudio previo. El objetivo principal de esta tarea es el de reducir la incertidumbre acerca del trabajo a realizar.

Configuración del entorno de trabajo

Antes de empezar a diseñar el sistema, se configurará el entorno de trabajo, que consistirá en la instalación del software necesario así como en la configuración pertinente del equipo.

Diseño del sistema monolítico

Teniendo el entorno de trabajo configurado, se procederá a realizar el diseño del sistema monolítico así como a generar toda la documentación necesaria para poder comenzar el desarrollo.

Desarrollo del sistema monolítico

El siguiente paso será desarrollar el sistema. Esto incluye la creación del proyecto y la implementación de las diferentes funcionalidades.

Puesta en producción

Para finalizar, deberá desplegarse el sistema en *AWS*. Para ello será necesario realizar la configuración de la infraestructura de contenedores, políticas de escalado, permisos, etc.

12.1.3 Análisis del rendimiento

Estudio previo

Antes de empezar, se realizará un pequeño estudio de las herramientas disponibles para poder realizar pruebas de estrés y se evaluará cual resulta mejor opción.

Configuración del entorno de trabajo

Una vez decidida la herramienta a utilizar, siempre y cuando se haga desde un entorno local, se procederá a instalarla y a configurar el equipo para poder realizar las pruebas. En caso de necesitar llevarlas a cabo desde alguna herramienta en la nube, se realizará la configuración de estas.

Realización de pruebas de carga

El objetivo de esta tarea será medir el rendimiento del sistema realizando peticiones masivas. Para ello se hará uso de las herramientas elegidas, simulando un uso intensivo comparable al que podría estar sometido un sistema de características similares en un entorno de producción con usuarios reales.

12.1.4 Análisis de los resultados obtenidos

A partir de las ventajas e inconvenientes identificados a lo largo del desarrollo y puesta en producción, así como de los resultados obtenidos mediante las pruebas de estrés, se realizará un análisis de ambas arquitecturas. Su objetivo será el de medir el desempeño de cada una de ellas para poder extraer las conclusiones del proyecto

12.1.5 Documentación final

Por último, será necesario redactar la memoria del TFG recogiendo los resultados obtenidos del proyecto, así como documentando todos los pasos seguidos en su desarrollo.

12.1.6 Preparación de la defensa

Una vez entregada la memoria, se invertirá todo el tiempo restante hasta el día de su defensa en la preparación de la misma.

12.2 Estimación de las tareas

A continuación se incluyen dos tablas: una con las estimaciones de cada una de las tareas y otra con sus dependencias.

ID	*Tarea*	*Duración estimada*
T1	**Documentación inicial**	**66h**
T2	**Sistema monolítico**	**108h**
T2.1	Estudio previo	20h
T2.2	Configuración del entorno de trabajo	2h
T2.3	Diseño del sistema monolítico	8h
T2.4	Desarrollo del sistema monolítico	40h
T2.5	Puesta en producción	38h
T3	**Sistema basado en microservicios**	**287h**
T3.1	Estudio previo	48h
T3.2	Configuración del entorno de trabajo	12h
T3.3	Diseño del sistema basado en microservicios	24h
T3.4	Desarrollo del sistema basado en microservicios	143h
T3.5	Puesta en producción	60h
T4	**Análisis del rendimiento**	**27h**
T4.1	Estudio previo	12h
T4.2	Configuración del entorno de trabajo	5h
T4.3	Realización de pruebas de estrés	10h
T5	**Análisis de los resultados obtenidos**	**22h**
T6	**Documentación final**	**32h**
T7	**Preparación de la defensa**	**24h**
TOTAL		**566h**

Tabla 1: Estimación de las tareas

ID	*Dependencias*
T2.4	T2.2, T2.3
T2.5	T2.4
T3.4	T3.2, T3.3
T3.5	T3.4
T4.3	T2.5, T3.5, T4.1, T4.2,
T5	T4.3
T7	T.6

Tabla 2: Dependencias entre tareas

12.3 Diagrama de Gantt

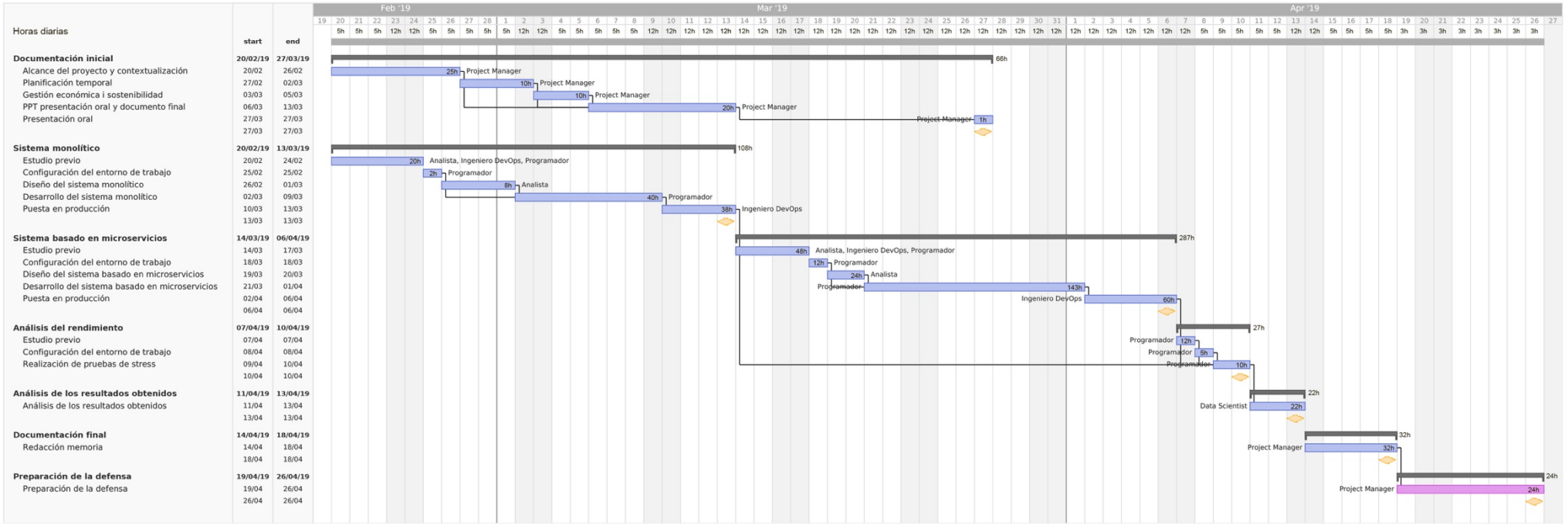


Ilustración 121: Diagrama de Gantt

12.4 Recursos

Recursos humanos

En la realización del proyecto, los diferentes roles expuestos en la Tabla 5 serán ejercidos por la misma persona.

También participarán tanto Ernest Teniente, director del trabajo, como Joan Subirats, profesor de GEP, que ayudarán a evaluar el trabajo realizado y a verificar que el rumbo de este es el correcto.

Recursos materiales

Para llevar a cabo el proyecto se utilizará un portátil *Macbook Pro* de 15 pulgadas para las tareas de desarrollo y redacción del trabajo.

También será necesario hacer uso de cierto software. Algunos ejemplos son el editor de código *IntelliJ IDEA*, el gestor de bases de datos *DataGrip*, el editor de textos *Microsoft Word* o el proveedor de servicios *AWS.*

Recursos físicos

Durante el transcurso del proyecto se hará uso de instalaciones de la Facultad de Informática de Barcelona tales como la biblioteca o las salas de estudio.

12.5 Plan de acción

Durante la planificación del proyecto, se identificaron posibles problemas que podrían surgir durante el transcurso del proyecto. A continuación se detallan las acciones a llevar a cabo en caso de darse alguno de estos.

12.5.1 Desarrollo del sistema

Al realizar la planificación del proyecto se tuvo en cuenta la falta de experiencia con alguna de las herramientas y tecnologías a utilizar. Por ese

motivo se creó una tarea de estudio previo al inicio de cada fase con el objetivo de ganar conocimientos.

Durante el desarrollo de ambos sistemas podrían aparecer errores o problemas que incrementen el tiempo necesario para llevarlo a cabo. La estimación actual ya contempla este tipo de sucesos, por lo que no debería haber contratiempos. Si, aun así, surgiera un imprevisto que conllevase un retraso significativo, se pasaría a prescindir de funcionalidades del sistema, lo cual reduciría el tiempo de desarrollo sin afectar a los resultados finales.

12.5.2 Puesta en producción

De las fases de desarrollo, la parte de puesta en producción es aquella con más incertidumbre y por lo tanto la más propensa a sufrir retrasos. Por ello han sido sobreestimadas con el fin de minimizar el impacto de estos en caso de ocurrir, especialmente con el sistema de microservicios.

En caso de surgir algún problema que pudiera poner en peligro el cumplimiento de las fechas establecidas, se recurriría a compañeros con altos conocimientos en tareas de *DevOps* para identificarlo y solucionarlo lo antes posible. En el peor de los casos se descartaría hacer las pruebas en la nube y se recurriría a realizarlas en local.

12.5.3 Análisis de rendimiento

La estimación de la fase de análisis de rendimiento es bastante ajustada, por lo que en caso de haber algún problema la duración total podría verse afectada. Con el fin de evitar posibles retrasos, se ha ido avanzando la tarea de estudio previo, así como realizando pruebas con diferentes herramientas durante la fase de desarrollo del sistema monolítico.

12.5.4 Necesidad de más recursos

Un retraso en las tareas de despliegue o análisis de rendimiento podría conllevar la utilización de recursos de *AWS* durante más tiempo de lo previsto, lo que significaría un aumento en los gastos.

Para controlar esto, se definirá un presupuesto máximo. Si este llegase a superarse se pasará a hacer uso de la capa gratuita, lo que conllevaría hacer uso de máquinas menos potentes para las pruebas.

12.6 Desviación temporal

Los tiempos expresados en la planificación temporal inicial son aproximaciones basadas en experiencias previas llevando a cabo labores similares.

Por ello, la duración real de las mismas no ha sido exactamente igual en la práctica, terminando algunas antes de lo previsto y teniendo que incrementar la dedicación en otras debido a problemas o imprevistos surgidos durante el desarrollo.

A continuación se detallan todas las desviaciones sufridas así como el impacto que han tenido en la planificación temporal final.

12.6.1 Modificaciones en las tareas

Las tareas definidas en la planificación inicial no han sufrido ninguna modificación a excepción de una, a la que se le añadió una sub-tarea más.

Implementación de tests

Debido a un aumento en el tiempo disponible para dedicar al proyecto, se añadió la tarea de implementar tests para probar el correcto funcionamiento del monolito. Esto incluye la implementación de tests unitarios y de integración haciendo uso de diversas bibliotecas.

Esta nueva tarea, con id T2.6 se llevó a cabo una vez finalizado el desarrollo de todas las funcionalidades.

12.6.2 Dedicación por tareas

En la siguiente tabla se especifica la estimación realizada al comienzo del proyecto y la dedicación real una vez finalizado.

ID	*Tarea*	*Duración estimada*	*Duración final*
T1	**Documentación inicial**	**66h**	**78h**
T2	**Sistema monolítico**	**108h**	**120h**
T2.1	Estudio previo	20h	20h
T2.2	Configuración del entorno de trabajo	2h	2h
T2.3	Diseño del sistema monolítico	8h	8h
T2.4	Desarrollo del sistema monolítico	40h	35h
T2.6	*Implementación de tests*	-	35h
T2.5	Puesta en producción	38h	20h
T3	**Sistema basado en microservicios**	**287h**	**244h**
T3.1	Estudio previo	48h	48h
T3.2	Configuración del entorno de trabajo	12h	12h
T3.3	Diseño del sistema basado en microservicios	24h	24h
T3.4	Desarrollo del sistema basado en microservicios	143h	85h
T3.5	Puesta en producción	60h	75h
T4	**Análisis del rendimiento**	**27h**	**58h**
T4.1	Estudio previo	12h	20h
T4.2	Configuración del entorno de trabajo	5h	10h
T4.3	Realización de pruebas de estrés	10h	28h
T5	**Análisis de los resultados obtenidos**	**22h**	**22h**
T6	**Documentación final**	**32h**	**102h**
T7	**Preparación de la defensa**	**24h**	**24h**
TOTAL			**648h**

Tabla 3: Dedicación real de las diferentes tareas del proyecto

Como se puede observar, hay pequeñas variaciones entre la estimación realizada y la dedicación final aunque el resultado es bastante similar, por lo que se entiende que las estimaciones eran acertadas. Sin embargo, hay cuatro tareas donde la diferencia es significativa:

- **(T2.5) Puesta en producción:** En un primer momento, se pensaba que el proceso de puesta en producción del monolito sería más costoso debido a la falta de experiencia con la plataforma. Por ello se estimó en 38 horas, aunque finalmente se ha realizado en 20.
- **(T3.4) Desarrollo del sistema basado en microservicios:** De nuevo, debido a la falta de experiencia con sistemas distribuidos y las herramientas necesarias para desarrollarlas, esta tarea se estimó en 143 horas, siendo la más costosa del trabajo.

Sin embargo, gracias a la reutilización del código del monolito, esta ha podido ser terminada en solo 85 horas, de las cuales gran parte se han destinado a la configuración de la infraestructura del sistema y de los contenedores.

- **(T4.3) Realización de pruebas de estrés:** Esta tarea no presentaba ninguna complejidad al principio, por lo que se estimó en 10 horas. Tiempo suficiente para poder familiarizarse con la herramienta y realizar las pruebas.

  No obstante, algunos problemas con los resultados obtenidos han hecho que la duración de esta se vea incrementada realizando pruebas adicionales.
- **(T6) Documentación final:** La tarea de redactar la memoria del proyecto ha sido la peor estimada de todas, ya que no se previó todo el trabajo necesario para explicar cada uno de los pasos seguidos, haciendo que las 32 horas de la estimación inicial se hayan convertido en 102.

12.6.3 Cambios en la planificación

Debido a un cambio en el turno de lectura para realizar la defensa del proyecto, la fecha de entrega del trabajo se cambió del 19 de Abril al 26 de Junio.

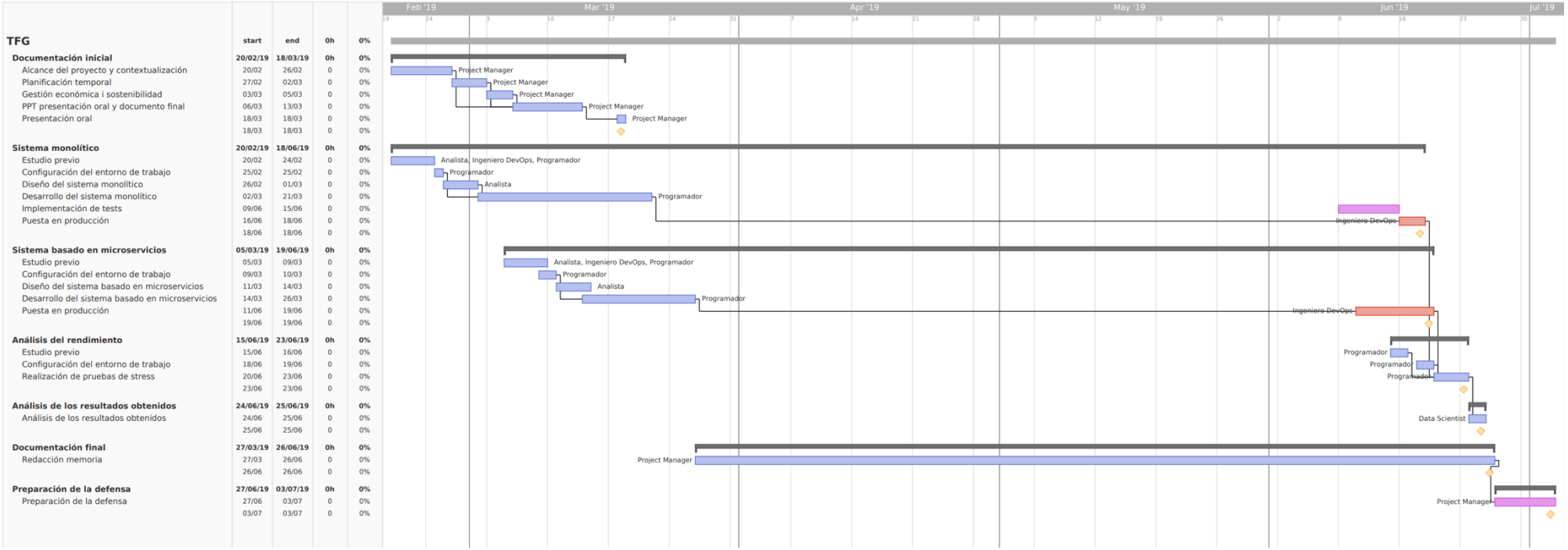
Este cambio afectó a la planificación inicial, ya que al disponer de más tiempo la distribución del mismo no se corresponde con la especificada en el diagrama Gantt anterior.

Por ello, en la siguiente tabla se muestran todas las tareas y las fechas en las que fueron realizadas finalmente.

ID	*Tarea*	*Fecha inicial*	*Fecha final*
T1	**Documentación inicial**	**20/02/2019**	**18/03/2019**
T2	**Sistema monolítico**	**20/02/2019**	**18/06/2019**
T2.1	Estudio previo	20/02/2019	24/02/2019
T2.2	Configuración del entorno de trabajo	25/02/2019	25/02/2019
T2.3	Diseño del sistema monolítico	26/02/2019	01/03/2019
T2.4	Desarrollo del sistema monolítico	02/03/2019	21/03/2019
T2.6	*Implementación de tests*	09/06/2019	15/06/2019
T2.5	Puesta en producción	22/03/2019	18/06/2019
T3	**Sistema basado en microservicios**	**05/03/2019**	**19/06/2019**
T3.1	Estudio previo	05/03/2019	09/03/2019
T3.2	Configuración del entorno de trabajo	09/03/2019	10/03/2019
T3.3	Diseño del sistema basado en microservicios	11/03/2019	14/03/2019
T3.4	Desarrollo del sistema basado en microservicios	14/03/2019	26/03/2019
T3.5	Puesta en producción	26/03/2019	19/06/2019
T4	**Análisis del rendimiento**	**16/06/2019**	**23/06/2019**
T4.1	Estudio previo	15/06/2019	16/06/2019
T4.2	Configuración del entorno de trabajo	18/06/2019	19/06/2019
T4.3	Realización de pruebas de estrés	20/06/2019	23/06/2019
T5	**Análisis de los resultados obtenidos**	**24/06/2019**	**25/06/2019**
T6	**Documentación final**	**27/03/2019**	**26/06/2019**
T7	**Preparación de la defensa**	**26/06/2019**	**03/07/2019**

Tabla 4: Cambios respecto a la planificación temporal inicial

12.6.4 Gantt corregido



13 Gestión económica

13.1 Identificación de costes

Con el objetivo de garantizar la viabilidad económica del proyecto, se realizó un análisis, de forma previa a su desarrollo, de todos los costes asociados al mismo.

En esta sección se detallan todos los costes identificados durante la planificación inicial del proyecto, tanto directos como indirectos, así como la desviación sufrida respecto a dicha planificación una vez finalizado.

13.1.1 Costes directos

La primera categoría de costes, los directos, se dividió en dos tipos diferentes: los asociados a recursos humanos y los asociados a recursos materiales.

A continuación se estima el impacto económico en el proyecto de cada uno de ellos.

Recursos humanos

El primer paso para cuantificar los costes de recursos humanos fue realizar una estimación de los salarios de cada uno de los roles involucrados en el proyecto, los cuales fueron añadidos al diagrama Gantt.

Para realizar una estimación precisa del coste por hora se consultaron datos en la web *PayScale [9]*, tomando como referencia el salario mediano, que se dividió entre las 1800 horas definidas en el convenio de consultoría.

ID	*Rol*	*€/hora*	*Horas*	*Salario*
R1	Project Manager	24.40€	122h	2976.80€
R2	Analista	16.60€	52h	863,20€
R3	Programador	15.00€	248h	3720.00€
R4	Ingeniero DevOps	19.40€	122h	2366.80€
R5	Data Scientist	18.80€	22h	413.60€
TOTAL			**566h**	**10340.40€**

Tabla 5: Coste total por roles

La siguiente tabla muestra la dedicación de cada uno de los roles para las diferentes tareas de las que consiste el proyecto.

Tarea	***Horas invertidas***					***Total***
	R1	**R2**	**R3**	**R4**	**R5**	
T1	66h	-	-	-	-	66h
T2	-	12h	50h	46h	-	108h
T3	-	40h	171h	76h	-	287h
T4	-	-	27h	-	-	27h
T5	-	-	-	-	22h	22h
T6	32h	-	-	-	-	32h
T7	24h	-	-	-	-	24h
TOTAL	**122h**	**52h**	**248h**	**122h**	**22h**	**566h**

Tabla 6: Horas invertidas por rol y tarea

Recursos materiales

i. Hardware

Para el desarrollo del proyecto era necesario hacer uso de un portátil y de un *smartphone,* ambos sin amortizar ya que se adquirieron en 2017 y 2019, respectivamente.

Para calcular la amortización de cada uno de los dispositivos se obtuvo el precio por hora de uso teniendo en cuenta su precio de compra y las horas de vida útil totales. Más tarde, este se multiplicó por las horas de utilización del mismo durante el proyecto.

Se da por hecho que un año tiene 220 días laborables y que una jornada es de 8 horas.

$$\frac{precio\ compra}{años\ de\ vida\ útil \times dias\ laborables \times horas\ jornada} \times horas\ uso\ TFG$$

Ilustración 122: Fórmula de amortización de recursos

Hardware	***Precio***	***Vida útil***	***Horas de uso***	***€/hora***	***Amortización***
Macbook Pro 15"	2799€	4 años	566	0.39759€	225.00€
Huawei P20 lite	259€	4 años	65 (1h/día)	0.03679€	2.41€
TOTAL					**227.41€**

Tabla 7: Costes derivados del uso de hardware

ii. Software

Para el desarrollo del proyecto también se requiere de cierto software. A continuación, se detallan los costes asociados a este.

Software	***Precio***	***Vida útil***	***Horas***	***Total***
Microsoft Word	71.17 €	1 año	122	4.93€
IntelliJ IDEA Ultimate	0.00€	-	-	0.00€
Postman	0.00€	-	-	0.00€
GitHub	0.00€	-	-	0.00€
GitKraken	0.00€	-	-	0.00€
Docker Desktop	0.00€	-	-	0.00€
TeamGantt	0.00€	-	-	0.00€
TOTAL				**4.93€**

Tabla 8: Costes derivados del uso de software

Con el fin de simplificar el cálculo de dichos costes, aquellos asociados al uso de *AWS* se mostrarán en una tabla aparte, ya que se paga por hora de uso.

Dado que el diseño del sistema de microservicios aún no se había realizado, en el momento de realizar la planificación inicial no se podía saber con exactitud cuáles serían los recursos necesarios. Se previó la creación de cinco microservicios, que, sumados al sistema monolítico, requerían seis balanceadores de carga y un máximo de 48 instancias EC2 (8 instancias por servicio).

Se estimó un máximo de 36 horas de uso de la infraestructura en *AWS*, que incluía tanto el despliegue del monolito y microservicios como la realización de las pruebas de rendimiento.

Servicio	*Unidades*	*€/hora*	*Horas*	*TOTAL*
Elastic Load Balancer	6	0.02€	36	4.32€
Instancias EC2 (t2.medium)	48	0.04€	36	70.84€
Amazon SNS	-	0,00€	-	0,00€
Amazon SQS	-	0.00€	-	0,00€
Amazon VPC	-	0,00€	-	0,00€
TOTAL				**75.16€**

Tabla 9: Costes derivados del uso de Amazon Web Services

13.1.2 Costes indirectos

i. Consumo eléctrico

Tanto los dispositivos utilizados para la realización del proyecto como las instalaciones donde se llevó a cabo tienen un consumo eléctrico cuyos costes también fueron contabilizados para el presupuesto.

Dispositivo	*Consumo*	*Horas de uso*	*Coste*
Macbook Pro	95W	566	7.53€
Smartphone	3W	65 (1h/día)	0.03€
Luces	15W	300	0.63€
TOTAL			**8.19€**

Tabla 10: Costes derivados del consumo eléctrico

Se asume que el precio por kWh es de 0.140€ y que aproximadamente la mitad del tiempo dedicado al proyecto se hizo con luces encendidas.

ii. Cuota internet

Debido a las tareas de investigación al principio de cada fase, así como del uso de un proveedor de servicios en la nube como *AWS* para desplegar los sistemas, el uso de internet es imprescindible. Dado que la estimación inicial de la duración del proyecto fue de dos meses y la cuota mensual es de 65€, el coste total asciende a 130€.

iii. Desplazamientos

Se estimó también un total de 10 trayectos Mataró-Barcelona para asistir a reuniones con el director de proyecto y realizar presentaciones, por lo que

el coste total aproximado de los desplazamientos equivale al de una T-10 de 3 zonas, es decir, 10.20€.

13.1.3 Costes de contingencia

Al realizar la planificación del proyecto, se asumieron ciertos riesgos que podrían afectar negativamente el desarrollo del mismo. Por ello, se asignó un presupuesto de contingencia en función del coste que conllevarían, así como de la probabilidad de que ocurrieran.

Riesgo aceptado	***Probabilidad***	***Coste***	***Coste total***
Retraso desarrollo	5%	4374.60€	218.73€
Retraso puesta en producción	20%	2779.80€	555.96€
Retraso análisis rendimiento	10%	647.00€	64.70€
Necesidad de más recursos en AWS	5%	75.10€	3.76€
TOTAL			**843.15€**

Tabla 11: Costes de contingencia

13.1.4 Costes de imprevistos

Se añadió también al presupuesto un margen de un 10% sobre el total de $CD + CI$ con el fin de reducir el impacto de cualquier imprevisto sobre el presupuesto realizado debido a errores en la planificación o descuidos y, por consiguiente, evitar una posible puesta en peligro del proyecto.

13.1.5 Costes totales

Para obtener los costes totales, se sumaron los costes directos, indirectos, de contingencia y de imprevistos obtenidos anteriormente.

Tipo	***Coste***
Costes directos	10647.89€
Costes indirectos	148.69€
Costes de contingencia	843.15€
Costes de imprevistos	1163.98€
TOTAL	**12803.71€**

Tabla 12: Costes totales del proyecto

En la siguiente tabla se desglosan los costes de cada una de las tareas en función de la categoría a la que pertenece:

Tarea	*Recursos Humanos*	*Recursos materiales*	*Costes indirectos*	*Costes de contingencia*	*Costes de imprevistos*	*Total*
T1	1610.40€	28.22€	17.34€	-	165.60€	1821.56€
T2	1841.60€	43.28€	28.37€	263.39€	217.66€	2394.30€
T3	4703.40€	114.45€	75.40€	511.30€	540.46€	5945.01€
T4	405.00€	86.23€	7.09€	68.46€	56.68€	623.46€
T5	413.60€	9.09€	5.78€	-	42.85€	471.32€
T6	780.80€	14.70€	8.41€	-	80.39€	884.30€
T7	585.60€	11.52€	6.30€	-	60.34€	663.76€
TOTAL	**10340.40€**	**307.49€**	**148.69€**	**843.15€**	**1163.98€**	**12803.71€**

Tabla 13: Costes totales desglosados por tareas

13.2 Control de costes

Durante la realización del proyecto podría haber desviaciones y, aunque la función de parte del presupuesto es la de cubrir los gastos que estas pudieran conllevar, se creyó conveniente contar con mecanismos para poder identificarlas.

Por ello, decidió realizarse un breve informe identificando posibles desviaciones, sus causas y el impacto que estas pudieran tener sobre la planificación al finalizar cada uno de los hitos marcados en el diagrama de Gantt. Para ello se ha hecho uso de los siguientes indicadores:

- Desviaciones en la realización de tareas (en horas):
  *(consumo estimado – consumo real) * coste real*
- Desviaciones en la realización de tareas (en coste):
  *(coste estimado – coste real) * consumo horas real*
- Desviaciones totales de recursos:
  (coste estimado total – coste real total)

13.3 Desviación económica

Como se observa en el apartado 12.6, la dedicación real de cada una de las tareas ha variado respecto a la planificación temporal realizada al inicio del proyecto. Además, durante el transcurso del mismo han surgido ciertos imprevistos que han causado un consumo mayor de recursos, tanto humanos como materiales.

Como consecuencia, la planificación económica también ha sufrido algunos cambios. A continuación se detalla toda la información sobre las desviaciones sufridas así como de los costes totales del proyecto una vez finalizado.

13.3.1 Costes directos

Recursos humanos

Debido al incremento de horas de dedicación necesarias para llevar a cabo el trabajo, los costes asociados al salario de los diferentes roles se ha visto incrementado, como se puede ver en las siguientes tablas.

ID	*Rol*	*€/hora*	*Horas*	*Salario*
R1	Project Manager	24.40€	204	4977.60€
R2	Analista	16.60€	54	896.40€
R3	Programador	15.00€	250	3750.00€
R4	Ingeniero DevOps	19.40€	118	2289.20€
R5	Data Scientist	18.80€	22	413.60€
TOTAL			**648h**	**12326.80€**

Tabla 14: Costes totales por rol (corregidos)

Tarea	*Horas invertidas*					*Total*
	R1	**R2**	**R3**	**R4**	**R5**	
T1	78h	-	-	-	-	78h
T2	-	14h	79h	27h	-	120h
T3	-	40h	113h	91h	-	244h
T4	-	-	58h	-	-	58h
T5	-	-	-	-	22h	22h
T6	102h	-	-	-	-	102h
T7	24h	-	-	-	-	24h
TOTAL	**204h**	**54h**	**250h**	**118h**	**22h**	**648h**

Tabla 15: Horas invertidas por rol y tarea (corregidas)

Recursos materiales

El incremento de horas no sólo ha afectado a los salarios de los diferentes roles, sino también a la amortización del hardware utilizado.

Hardware	*Precio*	*Vida útil*	*Horas de uso*	*€/hora*	*Amortización*
Macbook Pro 15”	2799€	4 años	648	0.39759€	257.63€
Huawei P20 lite	259€	4 años	68 (1h/día)	0.03679€	2.50€
TOTAL					**260.13€**

Tabla 16: Costes derivados del uso de hardware (corregidos)

Además también se han dado algunos imprevistos durante la tarea de puesta en producción que han impactado en el uso de recursos de AWS.

Servicio	***Unidades***	***€/hora***	***Horas***	***TOTAL***
Instancias EC2 (t2.medium)	8	0.0464€	168	7.79€
Instancias RDS (t2.medium)	5	0,0820€	168	13.78€
TOTAL				**21.57€**

Tabla 17: Costes derivados del uso de AWS (corregidos)

13.3.2 Costes indirectos

En cuanto a los costes indirectos, el incremento de la duración del trabajo ha afectado tanto a la cuota de internet, a la que se han sumado 65€ equivalentes a un mes adicional, como al consumo de los dispositivos, que también se ha visto incrementado.

Dispositivo	***Consumo***	***Horas de uso***	***Coste***
Macbook Pro	95W	648	8.62€
Smartphone	3W	68 (1h/día)	0.03€
Luces	15W	300	0.63€
TOTAL			**9.28€**

Tabla 18: Costes derivados del consumo eléctrico (corregidos)

13.3.3 Costes totales

Para finalizar, se han calculado de nuevo los costes totales. Esta vez no se han tenido en cuenta ni costes de contingencia ni costes de imprevistos, ya que estos no son necesarios debido a que el proyecto ya ha finalizado.

Tipo	***Coste***
Costes directos	12613.43€
Costes indirectos	214.48€
TOTAL	**12827.51€**

Tabla 19: Costes totales del proyecto (corregidos)

Tarea	*Recursos Humanos*	*Recursos materiales*	*Costes indirectos*	*Total*	*Total estimado*	*% error*
T1	1903.20€	40.95€	30.64€	**1974.79€**	1821.56€	8.41%
T2	1941.00€	40.95€	30.64€	**2012.59€**	2394.30€	-15.94%
T3	4124.60€	40.95€	30.64€	**4196.19€**	5945.01€	-29.41%
T4	870.00€	40.95€	30.64€	**941.59€**	623.46€	51.02%
T5	413.60€	40.95€	30.64€	**485.19€**	471.32€	2.94%
T6	2488.80€	40.95€	30.64€	**2560.39€**	884.30€	189.53%
T7	585.60€	40.95€	30.64€	**657.19€**	663.76€	-0.98%
TOTAL	**12326.80€**	**286.63€**	**214.48€**	12827.91€	**12803.71€**	**-0.19%**

Tabla 20: Costes totales desglosados por tareas (corregidos)

14 Sostenibilidad y compromiso social

14.1 Matriz de sostenibilidad

	PPP	Vida útil	Riesgos
Ambiental	Consumo de diseño 10/10	Huella ecológica 20/20	Ambientales 0/-20
Económico	Factura 8/10	Plan de viabilidad 10/20	Económicos -5/-20
Social	Impacto personal 10/10	Impacto social 10/20	Sociales -15/-20
Rango de sostenibilidad	28/30	40/60	-20/-60
	48		

Tabla 21: Matriz de sostenibilidad

14.2 Análisis de sostenibilidad

14.2.1 Económica

Dada la naturaleza de este trabajo, no resulta necesario realizar un estudio de mercado pues no se espera obtener rentabilidad y los sistemas desarrollados serán únicamente destinados a realizar un analizar el desempeño de diferentes arquitecturas.

Desde el punto de vista de un posible inversor, podría parecer que la viabilidad de este proyecto es muy baja, pues no se espera explotar los sistemas desarrollados para obtener beneficios. Sin embargo, el impacto económico que este podría tener es mucho mayor a la inversión que supone dado que los conocimientos adquiridos a raíz de este estudio pueden ayudar

a empresas a tomar mejores decisiones durante el diseño de sus plataformas, lo que podría suponer un gran ahorro a largo plazo.

Una gran parte del presupuesto para el proyecto se destina a cubrir los salarios de los diferentes roles. Estos se han calculado teniendo en cuenta perfiles con más de dos años de experiencia, con titulación universitaria y que vivieran en Barcelona, lo que ha hecho que el presupuesto se vea incrementado de forma significativa. En caso de ser necesario, se podrían buscar perfiles *junior,* lo cual permitiría reducir los costes totales del proyecto entre un 20% y un 30%, aproximadamente.

Debido a la planificación de costes realizada, donde ya se han tenido en cuenta posibles imprevistos y se ha añadido un presupuesto de contingencia, se considera altamente improbable que ocurra algo que incremente el impacto económico del proyecto.

14.2.2 Medioambiental

El impacto medioambiental de este trabajo es prácticamente nulo, ya que consiste principalmente en el consumo de un ordenador portátil, de un *smartphone* y de una bombilla LED de bajo consumo, en caso de trabajar de noche. Se estima, a partir de los consumos especificados en la Tabla 18, que el consumo energético total derivado de la realización de este trabajo es de 66.3 kWh que, a modo de comparación, es equivalente al consumo de un refrigerador de clase A durante ocho semanas y media.

También se ha tenido en cuenta el impacto derivado del uso de máquinas en la nube para alojar los dos sistemas desarrollados a lo largo del proyecto, y este es más bajo que si se utilizaran máquinas físicas ya que estas no son dedicadas y están optimizadas para ese uso concreto. Además, *Amazon Web Services* reduce la huella ecológica mediante el uso de energías renovables procedentes de sus propios parques eólicos y granjas solares.

Una gran parte del tiempo del proyecto se ha destinado al desarrollo de dos sistemas desde cero. Si en lugar de eso, se hubieran utilizado sistemas ya existentes y *opensource,* podría haberse el impacto medioambiental todavía

más, aunque no hubiera supuesto una gran diferencia pues este ya es muy bajo.

En cuanto a los riesgos, durante la planificación inicial no se identificó ninguno que pudiera tener un impacto significativo en este aspecto.

14.2.3 Social

Actualmente, muchas empresas deciden migrar sus sistemas a arquitecturas de microservicios, lo que supone una gran inversión de tiempo y de recursos. Tomar una buena decisión en el diseño de un sistema puede suponer un ahorro importante.

Mediante la realización de este trabajo se espera conseguir cifras acerca del rendimiento de dos arquitecturas, así como de sus ventajas e inconvenientes, que ayuden a mejorar la eficiencia de la toma de decisiones de aquellas empresas que actualmente se encuentren es esta situación. Esto podría suponer una reducción del tiempo necesario para llevar a cabo dicho proceso y, por consiguiente, un ahorro de recursos.

Aun así, existe el riesgo de que ninguna empresa llegue a hacer uso de los resultados obtenidos durante este estudio para tomar decisiones de diseño de sus sistemas. En ese caso, el impacto social del proyecto sería muy bajo.

Personalmente, la realización de este trabajo me aportará conocimientos en un tema de especial interés, como lo son las arquitecturas basadas en microservicios, y de las tecnologías necesarias para hacerlas funcionar. Los conocimientos adquiridos durante el desarrollo del proyecto también afectarán a mi actual empresa, que actualmente se encuentra en un proceso de migración del actual monolito a una arquitectura distribuida, por lo que se beneficiará directamente de todo lo que aprenda.

15 Conclusiones

Tras haber realizado el diseño e implementación de una red social haciendo uso de dos estilos arquitectónicos diferentes y medido su rendimiento mediante la realización de pruebas de carga, se introduce el último capítulo de la memoria.

Este tiene como objetivo analizar los resultados obtenidos, realizar una evaluación retrospectiva del trabajo realizado y decidir los siguientes pasos del proyecto.

15.1 Resultados obtenidos

A continuación se detallan todas las ventajas e inconvenientes encontrados durante el desarrollo de ambos sistemas, así cómo cuando deberían utilizarse a partir de la experiencia obtenida durante el desarrollo de este proyecto.

15.1.1 Sistema monolítico

Ventajas

- **Facilidad de desarrollo**: El desarrollo del monolito ha resultado sencillo ya que no requiere de ninguna infraestructura para ejecutarse, lo que ha permitido empezar a trabajar desde el primer momento sin invertir mucho tiempo en configuración.

- **Menor tiempo de desarrollo:** Debido a la simplicidad de la infrestructura necesaria para ejecutar el monolito, su tiempo de desarrollo ha sido significativamente menor que el del sistema de miroservicios.

- **Facilidad de despliegue:** Al ser un único artefacto, el proceso de despliegue del sistema monolítico ha sido sencillo e intuitivo, ya que apenas ha requerido ninguna configuración.

- **Buen rendimiento:** Gracias a que este sistema no requiere hacer peticiones a través de la red para comunicarse con diferentes servicios, realizar consultas resulta más rápido que un sistema distribuido.

- **Consistencia de los datos:** Como consecuencia de tener todos los datos en una sola base de datos, la consistencia de estos está asegurada, cosa que no es segura en sistemas distribuidos.

- **Menor coste:** El coste de desarrollar y poner en producción un sistema monolítico es significativamente más bajo que el de un sistema distribuido, ya que su desarrollo es más rápido y este se ejecuta en una única instancia, que aunque es más potente, resulta más económica que muchas con especificaciones más bajas.

Inconvenientes

- **Complejidad del sistema:** Durante el desarrollo del sistema monolítico se ha observado que a medida que se añadían nuevas funcionalidades, la complejidad de este aumentaba debido al gran número de ficheros y directorios. Esto podría resultar un problema a medida que el proyecto creciese.

- **Alto acoplamiento:** También se ha observado que muchas de las funcionalidades dependían de *Users,* por lo que, si no se es riguroso con la arquitectura del sistema pronto se encontrarían problemas de

acoplamiento que dificultarían futuros cambios, haciendo el mantenimiento más costoso.

¿Cuándo debería utilizarse?

Tras haber desarrollado y probado el sistema monolítico, se ha observado que este tipo de arquitecturas presentan un gran número de beneficios y sólo algunos problemas que, con cierto criterio a la hora de realizar el diseño y mucho cuidado a lo largo del tiempo, pueden ser evitados en gran medida.

A partir de los datos obtenidos y las observaciones realizadas, se cree que este tipo de arquitecturas resultan una buena opción para todo tipo de proyectos, aunque lo son especialmente para aquellos de tamaño medio y bajo.

También ha destacado destaca su menor coste de desarrollo, lo que las hace perfectas para empresas que no dispongan de un gran presupuesto.

A diferencia de los sistemas de microservicios, un sistema monolítico escala como un solo artefacto, por lo que si el sistema a desarrollar requiere de una gran demanda y se prevén picos en diferentes partes del sistema, es posible que este no resulte la mejor opción. Aunque se podría escalar sin problemas, la utilización de recursos sería poco eficiente.

15.1.2 Sistema de microservicios

Ventajas

- **Gran cambiabilidad:** Debido a que los servicios se encuentran totalmente desacoplados unos de otros y que si se hace un buen diseño la cohesión dentro de los mismos debería ser alta, realizar cambios es extremadamente fácil. Además cada uno de los servicios es significativamente más pequeño que un sistema monolítico.
- **Escalabilidad:** Aunque durante las pruebas de rendimiento no se ha llegado a escalar, es evidente que los sistemas basados en microservicios destacan por este factor. El hecho de poder escalar cada uno de los

diferentes servicios en lugar de hacerlo el sistema al completo es una gran ventaja.

- **Diversidad de tecnologías:** En un primer momento se pensó en utilizar diferentes bases de datos en cada uno de los servicios en función de la frecuencia y el tipo de acceso a los datos. Aunque finalmente no se llegó a hacer, hubiera sido muy fácil ya que cada servicio es totalmente independiente al resto, permitiendo el uso de las tecnologías más adecuadas.

Inconvenientes

- **Integridad de los datos:** Mantener la integridad de los datos, algo que se da por hecho en un sistema monolítico, ha sido todo un reto durante el diseño y desarrollo del sistema de microservicios. Para ello ha sido necesario implementar un sistema de mensajes asíncronos y aún así podría haber problemas al no utilizar operaciones transaccionales en los casos en que dos o más servicios estén involucrados.
- **Bajo rendimiento:** El rendimiento del sistema de microservicios ha sido significativamente más bajo que el del sistema monolítico. No se conoce con certeza el motivo que ha causado que esta diferencia sea tan grande, aunque es cierto que el hecho de tener que utilizar la red para realizar cualquier petición hace que estos tengan un rendimiento inferior.
- **Mayor tiempo de desarrollo:** Debido a la naturaleza del sistema de microservicios, es necesario configurar cierta infraestructura para que este funcione. Esto, junto con la necesidad de crear diversos servicios, ha hecho que el desarrollo sea más lento que el del sistema monolítico. Sin embargo, se cree que este último problema se iría atenuando con el tiempo, ya que el coste de añadir una nueva funcionalidad en un sistema depende en gran medida de la complejidad del mismo y en el caso de los microservicios la

complejidad de cad uno de ellos es mucho menor que en un sistema monolítico.

- **Dificultad de despliegue:** Este ha sido, con diferencia, el mayor reto a superar durante el proyecto. Poner en producción un sistema de microservicios, junto con toda su infraestructura y configurar políticas de auto escalado y balanceadores de carga ha sido considerablemente más complejo que en el sistema monolítico. Se confirma, por lo tanto, la gran dependencia de este tipo de arquitecturas con roles DevOps.
- **Mayor coste:** Debido a la infraestructura necesaria, al mayor número de servicios de un sistema de microservicios y a que cada uno de ellos cuenta con su propia base de datos, el coste de desplegar el sistema de microservicios ha sido notablemente más alto que el monolítico. Más concretamente, en el caso de este proyecto, el coste ha sido seis veces superior.

¿Cuándo debería utilizarse?

Tras ver todas las ventajas e inconvenientes encontrados durante los procesos de desarrollo y puesta en producción del sistema de microservicios, se ha llegado a la conclusión de que estos resultan una gran opción para grandes proyectos, que pueden crecer rápidamente con el tiempo y en los que trabaja un gran equipo de desarrolladores.

Esto último es importante, ya que cada equipo se puede hacer cargo del mantenimiento de diferentes servicios, haciendo que el conocimiento acerca de estos sea superior y mejorar así la eficiencia a la hora de realizar cambios.

Además, también es importante destacar que, debido a su elevado coste de desarrollo y mantenimiento, estos pueden no ser la opción adecuada para empresas pequeñas donde el presupuesto sea menor.

Por último, es importante tener en cuenta que las limitaciones de este tipo de arquitecturas, como la consistencia eventual, hacen que no sean los

más adecuados para cierto tipo de sistemas donde la integridad de los datos es de vital importancia, como por ejemplo el sistema de pagos de un banco.

15.2 Consecución de competencias

Uno de los primeros pasos a realizar al idear el proyecto, fue la elección de competencias que se pondrían en práctica durante el desarrollo del mismo. A continuación se detalla cada una de las competencias elegidas y una breve justificación de la consecución de cada una de ellas.

- **[Suficiente] CES1.1: Desarrollar, mantener y evaluar sistemas y servicios software complejos y / o críticos**

  Desarrollar, mantener y evaluar sistemas software ha sido la principal tarea en el transcurso del proyecto, ya que se han desarrollado dos sistemas diferentes, cada uno con tecnologías diferentes y se han evaluado con el fin de realizar una comparación.

- **[En profundidad] CES1.2: Dar solución a problemas de integración en función de las estrategias, de los estándares y de las tecnologías disponibles.**

  Para realizar el proyecto, especialmente en el caso del sistema basado en microservicios, se han utilizado estándares en el sector a la hora de desarrollar el sistema y las tecnologías mas populares en la actualidad para este tipo de proyectos.

- **[En profundidad] CES1.3: Identificar, evaluar y gestionar los riesgos potenciales asociados a la construcción de software que se pudieran presentar.**

  En las fases previas al diseño del sistema se realizó una planificación de las tareas a realizar, duración de cada una de ellas y posibles obstáculos que podrían surgir durante el transcurso del proyecto.

  Estos resultaron ser bastante acertados y permitieron evitar retrasos que hubieran afectado a la realización del trabajo dentro del tiempo establecido.

- **[Suficiente] CES1.5: Especificar, diseñar, implementar y evaluar bases de datos.**

  Aunque los sistemas a desarrollar no presentaban una gran complejidad, se realizó el diseño de las bases de datos que funcionarían tanto en el monolito como en los diferentes servicios.

  Además se realizó la configuración de las mismas tanto en un entorno local como en servicios en la nube.

- **[Suficiente] CES1.7: Controlar la calidad y diseñar pruebas en la producción de software.**

  Para verificar el correcto funcionamiento del sistema desarrollado se crearon diferentes tipos de pruebas automatizadas, unitarias y de integración, mediante el uso de diferentes herramientas.

- **[En profundidad] CES2.1: Definir y gestionar los requisitos de un sistema software.**

  A demás del desarrollo de ambos sistemas, la definición y gestión de requisitos de estos ha sido una de las principales tareas a realizar.

  De forma previa al desarrollo se definieron todos los requisitos funcionales que este debía cumplir y se especificaron diagramas de secuencia y el contrato de cada uno de ellos.

- **[Suficiente] CES2.2: Diseñar soluciones apropiadas en uno o más dominios de aplicación, utilizando métodos de ingeniería del software que integren aspectos éticos, sociales, legales y económicos.**

  Durante el diseño de ambos sistemas se tuvieron en cuenta diferentes aspectos, sobretodo el económico ay que era necesario medir el coste real del proyecto. También se tuvo en cuenta el impacto social que podrían tener los resultados obtenidos, ya que algunos actores involucrados se beneficiarán directamente de los mismos.

15.3 Adecuación a la especialidad

Este proyecto ha sido un ejercicio perfecto para poner en práctica diferentes conceptos tratados a lo largo de la especialidad de Ingeniería del Software.

Las asignaturas que más han influido en la realización de este trabajo han sido *Introducción a la Ingeniería de Software* y *Arquitectura de software*, que me han concienciado acerca de la importancia de realizar un buen diseño previo a la implementación de cualquier sistema.

También han sido de gran importancia las asignaturas en *Ingeniería de Requisitos,* que me ha ayudado a comprender la importancia de los pasos previos al desarrollo como especificar correctamente un proyecto, y *Aplicaciones y Servicios Web,* que fue una introducción al mundo de las aplicaciones y servicios web.

15.4 Trabajo futuro

El primer paso a seguir, ahora que el proyecto ha finalizado, será averiguar por qué los resultados obtenidos en las pruebas de rendimiento ha sido tan bajos, ya que en ambos casos el número de peticiones por segundo se encontraba por debajo del esperado. Además, hay muchas funcionalidades interesantes que, por falta de tiempo, no se han podido añadir al sistema.

Una de ellas es añadir autenticación *OAuth* en el *edge-service* que permita realizar peticiones al sistema únicamente a usuarios identificados.

Otro aspecto importante que ha quedado pendiente de realizar es la gestión de transacciones dentro del sistema distribuido, ya que en la actualidad no se gestionan de ninguna forma y en caso de error los datos incorrectos quedarían en el sistema.

Por último, también queda pendiente la realización de tests de contrato, que permitirían verificar que el contrato de las APIs de cada uno de los

servicios no se rompe al introducir un cambio y por lo tanto ayudarían a construir un sistema más seguro y fácil de cambiar.

15.5 Conclusiones finales

Una vez finalizado el proyecto y conociendo mejor el funcionamiento de cada uno de los estilos arquitectónicos con los que he trabajado, considero que no hay uno mejor que otro, sino que esto dependerá del tipo de sistema en el que se utilice.

Las arquitecturas de microservicios ofrecen una gran flexibilidad, pero a costa de un mayor coste de recursos. En contraste, los sistemas monolíticos pueden ser más rígidos y difíciles de mantener a la larga, pero ofrecen una serie de facilidades que los puede convertir en buen punto de partida para la mayoría de proyectos.

Bibliografía

[1] M. Fowler "*Microservices Resource Guide*" [Online] Disponible en: https://www.martinfowler.com/microservices/

[2] R. Annett. (19 de Noviembre de 2014) "*What is a Monolith?*" [Online] Disponible en: http://www.codingthearchitecture.com/2014/11/19/what_is_a_monolith.html

[3] M. Richards, "*Microservices vs. Service-Oriented Architecture*", 1st ed. O'Reilly Media Inc, 2016 [Online] Disponible en: https://learning.oreilly.com/library/view/microservices-vs-service-oriented/9781491975657/

[4] ItjobsWatch web, (20 de Febrero de 2019) [Online] Disponible en: https://www.itjobswatch.co.uk/jobs/uk/microservices.do

[5] C. Posta, P. Mińkowski, M. Eisele, M. McLarty, I. Nadareishvili, M. Makary, B. Scholl, C. Caldato, T. Jardinet "*DZone's guide to microservices: breaking down the monolith*", DZone [Online] Disponible en: https://dzone.com/guides/microservices-breaking-down-the-monolith

[6] J. Chen, R. R. Reilly, y G. S. Lynn, "*The Impacts of Speed-to-Market on New Product Success: The Moderating Effects of Uncertainty.*", 2005 [Online] Disponible en: https://ieeexplore.ieee.org/document/1424410

[7] S. Newman, "*Building Microservices*", 1st ed. O'Reilly Media Inc, 2018 [Online] Disponible en: https://learning.oreilly.com/library/view/building-microservices/9781491950340/

[8] K. Singh, M. Caliskan, O. Mihalyi and P. Pscheidl, "*Java EE 8 Microservices",* Packt Publishing, 2018 [Online] Displonible en: https://learning.oreilly.com/library/view/java-ee-8/9781788475143/

[9] PayScale web, (9 de Marzo de 2019) [Online] Disponible en: https://www.payscale.com/

[10] C. Posta (14 de Julio de 2016) "*The hardest part about microservices: Your data*" [Online] Disponible en: https://blog.christianposta.com/microservices/the-hardest-part-about-microservices-data/

[11] J. Lewis, P. Jamshidi, C. Pahl, S. Tilkov y N. C. Mendoça, "*Microservices: The Journey So Far and Challenges Ahead*", 2018 [Online] Disponible en: https://www.researchgate.net/publication/324959590_Microservices_The_Journey_So_Far_and_Challenges_Ahead

[12] P. Márton, (2017) *"Designing a Microservices Architecture for Failure"* [Online] Disponible en: https://blog.risingstack.com/designing-microservices-architecture-for-failure/

[13] D. Hubbell, (31 de Enero de 2018), "Top 4 Pros and Cons of Microservices Architecture" [Online] Disponible en: https://www.spkaa.com/blog/top-4-pros-cons-microservices-architecture/

[14] R. C. Martin, (13 de Agosto de 2012), "The Clean Architecture" [Online] Disponible en: https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html

[15] E. Alliaume, S. Roccaserra, (15 de Octubre de 2018), "Hexagonal Architecture: three principles and an implementation example" [Online] Disponible en: https://blog.octo.com/en/hexagonal-architecture-three-principles-and-an-implementation-example/

[16] OMG, (https://www.omg.org/spec/UML/2.4.1/Superstructure

[17] M. Pozzobon (22 de Abril de 2016), "Repository Pattern" [Online] Disponible en: http://codecleane.rs/2016/04/22/repository-pattern/

[18] E. Gamma, R. Helm, R. Johnson, J. Vlissides. "*Elements of reusable object-oriented software",* Addison-Wesley, 1994.

[19] Oracle, "*Java SE documentation",* [Online] Disponible en: https://docs.oracle.com/javase/7/docs/technotes/guides/language/

[20] E. Goebelbecker (11 de Diciembre de 2018) "*YAML Tutorial: Everything You Need to Get Started in Minutes*" [Online] Disponible en: https://rollout.io/blog/yaml-tutorial-everything-you-need-get-started/

[21] C. Richardson, "Microservices patterns", Manning, 2018, Chapter 8 [Online] Disponible en: https://learning.oreilly.com/library/view/microservices-patterns/9781617294549/kindle_split_016.html

[22] M. Fowler (6 de Marzo de 2014), "*Circuit Breaker*" [Online] Disponible en: https://martinfowler.com/bliki/CircuitBreaker.html

[23] Lalitha (12 de Noviembre de 2018), "*Service Registry and It's Patterns*", [Online] Disponible en: https://medium.com/@lalitham/microservices-service-registry-and-its-patterns-35d725cbfb67

[24] J. Lim (7 de Noviembre de 2017), "*Best Practices for Response Times and Latency*", [Online] Disponible en: https://github.com/Tendrl/documentation/wiki/Best-Practices-for-Response-Times-and-Latency